THE UNIVERSITY *of* EDINBURGH

A Dynamically Reconfigurable Asynchronous Processor

Khodor Ahmad Fawaz

Doctor of Philosophy

The University of Edinburgh

March 2012

Declaration

This thesis entitled “A Dynamically Reconfigurable Asynchronous Processor” is submitted to the University of Edinburgh for the degree of Doctor of Philosophy. The research work described and reported in this thesis has been completed solely by Khodor Ahmad Fawaz under the supervision of Professor Tughrul Arslan. Where other sources are quoted, full references are given.

Khodor Ahmad Fawaz
March 2012

Abstract

The main design requirements for today's mobile applications are:

- high throughput performance.
- high energy efficiency.
- high programmability.

Until now, the choice of platform has often been limited to Application-Specific Integrated Circuits (ASICs), due to their best-of-breed performance and power consumption. The economies of scale possible with these high-volume markets have traditionally been able to hide the high Non-Recurring Engineering (NRE) costs required for designing and fabricating new ASICs. However, with the NREs and design time escalating with each generation of mobile applications, this practice may be reaching its limit.

Designers today are looking at programmable solutions, so that they can respond more rapidly to changes in the market and spread costs over several generations of mobile applications. However, there have been few feasible alternatives to ASICs: Digital Signals Processors (DSPs) and microprocessors cannot meet the throughput requirements, whereas Field-Programmable Gate Arrays (FPGAs) require too much area and power.

Coarse-grained dynamically reconfigurable architectures offer better solutions for high throughput applications, when power and area considerations are taken into account. One promising example is the Reconfigurable Instruction Cell Array (RICA). RICA consists of an array of cells with an interconnect that can be dynamically reconfigured on every cycle. This allows quite complex datapaths to be rendered onto the fabric and executed in a single configuration - making these architectures particularly suitable to stream processing. Furthermore, RICA can be programmed from C, making it a good fit with existing design methodologies.

However the RICA architecture has a drawback: poor scalability in terms of area and power. As the core gets bigger, the number of sequential elements in the array must be increased significantly to maintain the ability to achieve high throughputs through pipelining. As a result, a larger clock tree is required to synchronise the increased number of sequential elements. The clock tree therefore takes up a larger percentage of the area and power consumption of the core.

This thesis presents a novel Dynamically Reconfigurable Asynchronous Processor (DRAP), aimed at high-throughput mobile applications. DRAP is based on the RICA architecture, but uses asynchronous design techniques - methods of designing digital systems without clocks. The absence of a global clock signal makes DRAP more scalable in terms of power and area overhead than its synchronous counterpart.

The DRAP architecture maintains most of the benefits of custom asynchronous design, whilst also providing programmability via conventional high-level languages. Results show that the DRAP processor delivers considerably lower power consumption when compared to a market-leading Very Long Instruction Word (VLIW) processor and a low-power ARM processor. For example, DRAP resulted in a reduction in power consumption of 20 times compared to the ARM7 processor, and 29 times compared to the TIC64x VLIW, when running the same benchmark capped to the same throughput and for the same process technology (0.13µm). When compared to an equivalent RICA design, DRAP was up to 22% larger than RICA but resulted in a power reduction of up to 1.9 times. It was also capable of achieving up to 2.8 times higher throughputs than RICA for the same benchmarks.

In the name of Allah, the Beneficent, the Merciful.

Read: In the name of thy Lord Who createth, Createth man from a clot. Read: And thy Lord is the Most Bounteous, Who teacheth by the pen, Teacheth man that which he knew not. Nay, but verily man is rebellious That he thinketh himself independent! Lo! unto thy Lord is the return.

The Quran; Chapter 96, Verses 1–7

Dedicated to my father Mr Ahmad Fawaz: Thank you for always believing in me and supporting me.

Acknowledgements

I would like to thank my supervisors Professor Tughrul Arslan and Dr Iain Lindsay for their excellent supervision, expert guidance, friendship and support in this research endeavour not to mention their motivational enthusiasm and unconditional help. I would like to thank my friends Dr Sami Khawam, Dr Mark Muir, and Dr Ioannis Nousias for their valuable guidance and help throughout the project. I would also like to thank the following: Mical Johnstone for his help with editing the thesis, Eenass Natafji for keeping me well fed, and Dr. Ahmet Erdogan and Dr. Aristides Efthymiou for their support and help throughout the work.

I would like to thank the Overseas Research Student Awards Scheme (ORSAS) and the Edinburgh University's Institute for Integrated Micro and Nano Systems (IMNS) for funding this research.

Special thanks to my best friend and mila, Lidija Inara Liegis for her continuous support and encouragement throughout my PhD and for enduring the tedious task of proof-reading this thesis several times.

I would like to thank my parents, Ahmad and Faten, my sisters, Haifa, Nadine and Zeinab, and my nieces, Layla and Marya and my nephews, Ahmad and Adam for their amazing patience, encouragement and support without which this study would not be possible. To them I dedicate this thesis.

Finally, I would like to thank our God Almighty who has continuously guided me in all my endeavours.

Contents

Chapter 1: Introduction 1

1.1 Objectives and Scope of Research 4

1.2 Contribution to Knowledge 5

1.3 Publications 6

1.4 Structure of Thesis 6

Chapter 2: Literature Review 8

2.1 Asynchronous Circuit Design 8

2.1.1 Definition and Basic Concepts 9

2.1.2 Handshaking 12

2.1.3 Signalling Protocols 13

2.1.4 Muller C-element 15

2.1.5 Data Encoding 16

2.1.6 Completion Detection 18

2.1.7 Operation Modes 19

2.1.8 Delay Models 20

2.1.9 Muller and Huffman Models 22

2.1.10 Specification and Synthesis of Control Circuits 23

2.1.11 Asynchronous Datapaths 25

2.1.12 Testing and Synthesis for Testability 26

2.2 Requirements for Mobile Applications 26

2.2.1 High Performance and Energy Efficiency 27

2.2.2 Flexibility and Programmability 27

2.3 Reconfigurable Computers and Dynamic Reconfigurability 28

2.3.1 Microprocessors 29

2.3.2 Reconfigurable Computers 29
2.4 Asynchronous Reconfigurable Architectures 32
2.4.1 Asynchronous FPGAs 34
2.4.2 Coarse-grained Asynchronous Reconfigurable Computers 36
2.5 The RICA Architecture 37
2.6 Summary 38
Chapter 3: DRAP Overview 40
3.1 Architecture overview 42
3.2 Characteristics of DRAP 45
3.2.1 Energy driven consumption 45
3.2.2 Implicit Pipelining 48
3.2.3 Scalability 48
3.2.4 Reduced program size 49
3.3 Summary 50
Chapter 4: Asynchronous Operational Cells 52
4.1 Overview of Cell Design for DRAP 52
4.2 Synthesis of Asynchronous Operational cells 54
4.3 Cell Design Variations 57
4.4 Description of the Operational Cells 59
4.5 Summary 62
Chapter 5: Asynchronous Interconnect Design 63
5.1 DRAP Interconnect 63
5.2 The Routing Switch 68
5.2.1 Multi-Channel Interconnect Design 71
5.2.2 Incorporating Cells in Routing Switch 72

5.3 Challenges of Interconnect Design for Asynchronous Reconfigurable Circuits 73
5.4 Common Design Techniques 76
5.4.1 Overview 76
5.4.2 Techniques for Conditional Acknowledge Synchronisation 78
5.5 Proposed Technique for Acknowledge Synchronisation 81
5.6 An Interconnect Design Comparison 85
5.6.1 Proposed vs. Traditional Methods 88
5.6.2 Pipelined vs. Non-pipelined Switchboxes 89
5.7 Summary 90
Chapter 6: Controlling Reconfigurability in DRAP 91
6.1 Overview 92
6.2 ARC module design 93
6.2.1 Extracting the FINISH signal 94
6.2.2 Improvements 95
6.3 Scalability of technique 98
6.4 Jump cell 101
6.5 Summary 103
Chapter 7: Automatic Tool Flow 104
7.1 Implicit and Explicit Pipelining 104
7.1.1 Explicit Pipelining 105
7.1.2 Implicit Pipelining 105
7.1.3 Explicit vs. Implicit Pipelining 105
7.1.4 Non-pipelined Datapath on RICA vs. DRAP 106
7.1.5 Pipelined Datapath on RICA vs. DRAP 108

7.1.6 Limitations of Implicit Pipelining 109
7.2 Description of the Tool Flow 111
7.3 Summary 115
Chapter 8: Implementations on DRAP 117
8.1 Description of Example Algorithms 118
8.1.1 Bilinear Demosaic Filter 118
8.1.2 FFT 121
8.2 Description of Sample Arrays 124
8.2.1 20x20, 1-Channel DRAP 124
8.2.2 15x15, 5-Channel DRAP 125
8.2.3 Comparing DRAP with Other Architectures 125
8.3 Results 126
8.3.1 Pipelined vs. Non-pipelined DRAP 126
8.3.2 Single-channel vs. Multi-channel DRAP 132
8.3.3 Bilinear Demosaic on 15x15 DRAP 134
8.3.4 FFT on 15x15 DRAP 137
8.3.5 DRAP vs. Other Architectures 141
8.4 Summary 145
Chapter 9: Conclusions 148
9.1 DRAP Features 150
9.1.1 Scalability 150
9.1.2 Implicit Pipelining 151
9.1.3 Reduced Program Size 152
9.1.4 Event-driven Energy Consumption 154
9.2 Future Direction 156
9.2.1 Software 156

9.2.2 Hardware 157
9.2.3 Analysis 158
Appendix A: DRAP Cells and their Operations 173
A.1 ADDCOMP 173
A.2 JUMP 174
A.3 LOGIC 174
A.4 MUL 175
A.5 REG 175
A.6 SBUF 176
A.7 SHIFT 177
A.8 SINK 177
A.9 SOURCE 178
Appendix B: The DVB-SH Standard 179
Appendix C: Haste and the TiDE Tool Flow 181
C.1 Overview 181
C.2 Haste Notations 183
Appendix D: C Source Codes 184
D.1 Bilinear Demosaic – C Source Code 184
D.2 FFT – C Source Code 193

List of figures

Figure 2.1: Synchronous circuit block diagram [14]. 9

Figure 2.2: Handshaking: control and data (push and pull) channels [16]. 13

Figure 2.3: Two-phase asynchronous signalling protocol [13]..... 14

Figure 2.4: Four-phase asynchronous signalling protocol showing an early scheme [13]. ... 14

Figure 2.5: Symbol (left) and gate-level implementation of a Muller C-element (right). 16

Figure 2.6: Bundled data encoding - each bit is represented by one wire; one wire (request/acknowledge) to indicate validity..... 17

Figure 2.7: Dual-rail encoding - each bit is represented by two wires; the target in this case has to check for the validity of the received data bits before using them or asserting the acknowledge signal. 17

Figure 2.8: Comparison of delay models - redundancy required to eliminate hazards versus locality of timing assumptions [34]..... 20

Figure 2.9: Position of reconfigurable computing (adapted from [68])..... 30

Figure 2.10: Overview of the topology used in the asynchronous FPGA of [86]..... 35

Figure 2.11: Simplified example of a RICA based architecture. 37

Figure 3.1: Forecast for executing DSP algorithms on various architectures..... 41

Figure 3.2: Overview of the DRAP architecture. R0 to R3 are asynchronous register cells. The cells in this figure are distributed randomly..... 42

Figure 3.3: Software flow for programming DRAP starting from high-level C program. 45

Figure 3.4: Invalid data causing unnecessary activity in the combinatorial cells of a synchronous design, leading to wasted energy..... 46

Figure 4.1: Asynchronous circuit block diagram showing the Control and Datapath parts. . 55

Figure 4.2: (a) Haste description of an operational cell (b) Equivalent handshake circuit (c) A closer look at the control signals inside a handshake component: the C-element

synchronises the incoming acknowledge signals and the resulting signal passes through a delay matching the critical path of the function. 56

Figure 5.1: A crossbar interconnect scheme using multiplexers [81]. 64

Figure 5.2: Array of operational cells in an island-style mesh-based topology. 66

Figure 5.3: The switchbox interconnect with the operational cell inside..... 67

Figure 5.4: Combinatorial design of a DRAP routing switch. 68

Figure 5.5: Interconnect delays in array using combinatorial interconnects..... 69

Figure 5.6: Asynchronous design of a routing switch..... 70

Figure 5.7: Interconnect delays in array using pipelined interconnects. 71

Figure 5.8: An asynchronous sender connecting to multiple receivers..... 74

Figure 5.9: A cell (X) connected to its four neighbours using a C-element to synchronise the acknowledge signal: when the cell is programmed to connect to all four neighbouring cells (top configuration), the C-element synchronises all acknowledge signals after receiving them and correct communication occurs. When the cell is programmed to connect to only two neighbouring cells, E and S (bottom configuration), the C-element is waiting for an acknowledge signal from the N and W cells. These are not programmed to arrive hence communication between the cells is at a deadlock. 75

Figure 5.10: (a) Pipelined copy stage, which includes data latches controlled by the handshaking signals – the acknowledge signals are synchronised within the asynchronous control. (b) Non-pipelined copy stage where data and request are passed through and acknowledge signals are synchronised separately. 77

Figure 5.11: Pipelined copy stage used by the asynchronous reconfigurable architectures in [86]. Each cell connects to its four neighbours. A possible implementation of a programmable C-element is also shown (only control part of the circuit is shown). 78

Figure 5.12: Pipelined copy stage used by asynchronous reconfigurable architecture in [87]. Each cell is limited to connect to two other cells (only control part is shown). ... 80

Figure 5.13: Overview of the acknowledge synchronising method. 81

Figure 5.14: Ack-encode truth table. The select signals identifies if the ack signal is active or not. If it is, then the ack signal is routed through to ack-‘0’ and ack-‘1’. Otherwise, ack-‘0’ is set to logic 0 and ack-‘1’ is set to logic 1. 82

Figure 5.15: (a) Ack-encode of a receiver (R1), which is connected to two senders (S1, S2). (b) Ack-combine of a sender (S1) which connects to three receivers (R1, R2, R3). 83

Figure 5.16: Block diagram of an operational cell and switchbox using the proposed technique to synchronise acknowledge signals. 84

Figure 5.17: An optimisation applied to copy stage design of Figure 5.11 (The Ack-encode are at the receiver side). 85

Figure 5.18: Normalised area and power consumption of the designs. 88

Figure 5.19: Plot showing how the total interconnect delay (Total_ID), acknowledge interconnect delay (A_ID), and data-request interconnect delay (D_R_ID) vary with the number of switchboxes connecting two cells. 89

Figure 6.1: Overview of how reconfiguration is controlled in DRAP. The Asynchronous Reconfiguration Controller (ARC) interacts with certain cells in the array - the JUMP and endpoint (EP) cells - to indicate when a step has finished its work.... 93

Figure 6.2: A closer look at how the Asynchronous Reconfiguration Controller (ARC) interacts with the CJUMP and endpoint (EP) cells. 94

Figure 6.3: A closer look at the optimised Asynchronous Reconfiguration Controller (ARC) and its environment. The AND gate delay is reduced and the ARC uses the early_cell_DONE from the endpoint (EP) cells. 96

Figure 6.4: Extracting cell_DONE and early_cell_DONE signals for the endpoint (EP) cells. 97

Figure 6.5: An example of a configuration with four data-paths. The routing algorithm is required to ensure that the pre-routing critical path remains the critical path after routing and ends in one of the end-point cells. This adds significant pressure to the routing algorithm. 99

Figure 6.6: The routing algorithm simply chooses any of the datapaths in a configuration context and artificially extends it to become the critical path and terminate at an endpoint cell. With this method, the additional pressure on the routing algorithm is removed and both hardware and software are simplified. Additionally this method is scalable as the number of endpoint cells need not increase as the array increases. 100

Figure 6.7: Block diagram of the asynchronous CJUMP cell. 102

Figure 6.8: Overview of how the CJUMP cell interacts with its environment. 103

Figure 7.1: Block diagram of an asynchronous cell with latches at the input channels. 106

Figure 7.2: DRAP vs. RICA datapath with no explicit pipelining. 107

Figure 7.3: DRAP vs. RICA datapath with three-stage explicit pipelining. 108

Figure 7.4: (a) Data flow graph of a bypass datapath. (b) Data flow graph of a bypass datapath with explicit pipelining added to the bypass branch to achieve full pipelining. 110

Figure 7.5: Automatic tool flow for DRAP. 112

Figure 7.6: Routing modification for DRAP. After initial allocate and route, the routed netlists undergo an optimising process in order to get the endpoint cells connected to the ARC. 114

Figure 8.1: The bilinear demosaic filter kernel data flow graph. 119

Figure 8.2: Radix-2 complex butterfly computation. 122

Figure 8.3: The radix-2 FFT butterfly kernel data flow graph. 122

Figure 8.4: Normalised throughput, power and energy consumption graph of the bilinear demosaic benchmark on a pipelined and non-pipelined 20x20, 1-channel DRAP. 127

Figure 8.5: Breakdown of the total area of the 20x20, 1-channel array with both non-pipelined and pipelined interconnect as percentage of cell and switchbox (Sbox) area. The total area of the array with pipelined interconnect is 33% larger than the array with non-pipelined interconnect. 131

Figure 8.6: Normalised throughput, power and energy consumption graph of the bilinear demosaic benchmark on a 15x5, 5-channel pipelined DRAP array and a 20x20, 1-channel pipelined one. 134

Figure 8.7: Breakdown of the total power consumption of the bilinear demosaic benchmark on the 15x15, 5-channel pipelined DRAP array, as a percentage of leakage and dynamic power. 135

Figure 8.8: Breakdown of the dynamic power consumption of the bilinear demosaic benchmark on the 15x15, 5-channel pipelined DRAP array, as a percentage of internal and switching power. 136

Figure 8.9: Breakdown of the total power consumption of the bilinear demosaic benchmark on the 15x15, 5-channel pipelined DRAP array, as a percentage of cell and switchbox (Sbox) power. 136

Figure 8.10: Breakdown of the total power consumption of the 8K radix-2 FFT benchmark on the 15x15, 5-channel pipelined DRAP array, as a percentage of leakage and dynamic power. 137

Figure 8.11: Breakdown of the dynamic power consumption of the 8K radix-2 FFT benchmark on the 15x15, 5-channel pipelined DRAP array, as a percentage of internal and switching power. 138

Figure 8.12: Breakdown of the total power consumption of the 8K radix-2 FFT benchmark on the 15x15, 5-channel pipelined DRAP array, as a percentage of cell and switchbox (Sbox) power. 138

Figure 8.13: Normalised power consumption (with optimised pipelining and matching throughput) of the benchmarks on DRAP, RICA_225, and ASIC. 144

Figure 8.14: Normalised power consumption (with optimised pipelining and matching throughput) of the benchmarks on DRAP, ARM7, and TIC64x. 144

List of tables

Table 2.1: Categorisation of asynchronous reconfigurable computers. 33

Table 3.1: Area comparison of cells with asynchronous control (0.13µm process technology). 46

Table 3.2: Comparing area of interconnects and number of configuration bits for asynchronous (DRAP) and synchronous (RICA) designs. 47

Table 4.1: Possible operational cells and their supported operations. 54

Table 4.2: Comparing area, delay, power, and energy of different versions of an asynchronous operational cell (ADDCOMP). 58

Table 5.1: Comparison between cross-bar and island-style interconnects. [72] 65

Table 5.2: Comparison of the number of configuration bits, normalised gate area and power consumption of the switchboxes designed both the proposed technique and the traditionally used technique. 87

Table 5.3: Minimum Interconnect delay introduced by using the different acknowledge synchronising methods in a non-pipelined switchbox. The delay is based on connecting the output of a cell to the input of its neighbouring cell. 87

Table 5.4: Fixed interconnect delay introduced by using the different acknowledge synchronising methods in a pipelined switchbox. 87

Table 8.1: Maximum instance counts of the cells for the bilinear demosaic filter. 120

Table 8.2: Maximum instance counts of the cells for the radix-2 FFT butterfly. 123

Table 8.3: Asynchronous operational cells in 20x20 arrays. 124

Table 8.4: Asynchronous operational cells in 15x15 arrays. 125

Table 8.5: Bilinear demosaicing (1 line x 2056 pixels) on 20x20, 1-channel 32 bit array, pipelined vs. non-pipelined interconnect. 130

Table 8.6: Bilinear demosaicing (1 line x 2056 pixels) on 20x20, 1-channel 18 bit pipelined array, pipelined vs. on 15x15, 5-channel 18 bit pipelined array. 133

Table 8.7: Bilinear demosaicing (1 line x 2056 pixels) on 15x15, 5-channel 18 bit array pipelined: dynamic and leakage power consumption. 133

Table 8.8: 8K radix-2 FFT on 15x15, 5-channel 18 bit array pipelined: dynamic and leakage power consumption. 140

Table 8.9: 8K radix-2 FFT on 15x15, 5-channel 18 bit array pipelined: cell and switchbox (Sbox) power consumption..... 140

Table 8.10: Bilinear demosaicing on 15x5, 5-channel 18-bit array with pipelined interconnect running 1 line of 2056 pixels. 143

Table 8.11: 8K radix-2 FFT on 15x5, 5-channel 18-bit array with pipelined interconnect. 143

Table 8.12: DRAP and RICA_225 array area and number of configuration bits. 143

Table 8.13: The effect of using pipelined and multi-channel interconnects on DRAP. 146

Table 8.14: Comparing DRAP to RICA and other leading technologies. 147

Table 9.1: A summary of DRAP vs. RICA and DRAP vs. DSP/VLIW processors. 155

Table 9.2: A summary of DRAP vs. ASIC. 155

Nomenclature

Abstract netlist	A netlist describing only which connections exist, but not how those connections are mapped onto the interconnect. This means the timing information is not accurate.
Assembly	Assembly language (also loosely known as assembler). A low-level but human-readable language which directly corresponds to instructions in the target architecture's instruction set. Assembly provides a thin layer of abstraction above machine language, where the instructions and operands would be coded directly in binary.
Asynchronous logic	A method of designing digital systems without clocks.
Bundled data	A method of encoding data in asynchronous circuit design, where one wire represents each bit. There is a separate wire which indicates validity of the entire data.
Arity	The number of arguments or operands that the function takes.
C-element	A state holding element used to synchronise events in asynchronous circuits.
Channel	The medium upon which two asynchronous sub-systems communicate by handshaking. This includes the control channel (request and acknowledge signals) and the data channel.
Coarse-grained	Refers to the width of the interconnect of a reconfigurable architecture. Functional units for coarse-grained architectures perform word-level operations.
Compiler	A software tool that converts source code from a high-level programming language (e.g. C) into a lower-level language to be interpreted by a machine.
Configuration context	The data describing the configuration of a reconfigurable architecture's functional units and interconnect at a given moment in time, in order for it to form a specific set of datapaths. Reconfiguration consists of loading a new configuration context. Also referred to as a wide instruction.
Data flow graph	A graph describing how operations in a basic block are connected together into datapaths. The nodes are operations mapping to cells, and the edges indicate data dependencies between the operations.

Datapath	A chain of connected operations, where the result of one operation is used as an input to one or more dependent operations in the chain. Operations belong to the same datapath if there is at least one unbroken path connecting them involving any number of other operations in between.
Dual rail	A method of encoding data in asynchronous circuit design, where two wires represent each bit.
Endpoint	A name given to the last operational cell in a datapath.
Explicit pipelining	Pipelining provided by using register cells.
Fine-grained	Refers to the native width of the interconnect of a reconfigurable architecture. Functional units for fine-grained architectures perform bit-level operations.
Functional units	The elements of a hardware array which operate on data. These can be logic gates, ALUs, or even processor cores.
Glitch	A glitch is a result of a non-monotonic transition between one voltage level and the other.
Handshaking	A signalling scheme used by most asynchronous circuits. A handshake involves the use of two basic control signals: a request signal, sent from an initiator to a target, which initiates an action and a corresponding acknowledge signal, from the target to the initiator, which signals the completion of that particular action.
Haste	A CSP programming language used by the TiDE tool to design asynchronous circuits.
Hazard	A circuit containing an output which may glitch.
Implicit pipelining	Pipelining provided by latches/flip-flops found inherently in asynchronous circuits.
Operational cell	The name given to the functional units of DRAP, where the functional units correspond in functionality to instructions common in RISC instruction sets.
Kernel	An inner loop in a compute-intensive application, which normally runs for many consecutive iterations. In DRAP and RICA, this term is also used to refer to a subset of these where the loop body can fit entirely within a single configuration context. This is the most efficient way to

	execute, as the configuration context only has to be loaded once during the run time of the loop.
Line buffer	Another name for a stream memory, used in the context of image signal processing, where local storage is normally for lines of the image near the line currently being processed.
Mapper	A software tool that determines how paths should be rendered onto the reconfigurable interconnect of a particular architecture, in order to achieve the connections described in an abstract netlist, without conflicts. The output is a routed netlist.
Netlist	A file describing the connectivity between functional units in a reconfigurable architecture. The file describes the graph for each configuration context, where the nodes are the functional units (cells), and the edges are the connections. Edges can contain properties that describe the path taken along the interconnect.
Non-return-to zero	A signalling protocol for asynchronous circuits. Also known as two-phase signalling.
Program counter	A register that controls which instruction/configuration context is to be executed.
Reconfigurable	A term given to computing architectures that are not hardwired to perform a single function—i.e. they can change the shape of their datapaths in order to change the functionality of the device. The hardware consists of functional units and interconnect (called a fabric), on top of which datapaths are rendered.
Return-to zero	A signalling protocol for asynchronous circuits. Also known as four-phase signalling.
Routed netlist	A netlist augmented with path information, showing how each of the connections are physically realised on the reconfigurable interconnect of the target architecture.
Scheduler	A software tool that converts the basic blocks of a linear assembly into parallel datapaths that are to be rendered onto a reconfigurable architecture. In general, a scheduler extracts parallelism from a sequential stream of operations.
Step	Another name for configuration context.

Stream memory	Local on-chip random-access memory used as local storage in high-bandwidth streaming applications. Stream memory is normally partitioned into multiple banks to increase the bandwidth.

Acronyms

3G	Third generation mobile telecommunications.
4G	Fourth generation mobile telecommunications.
A_ID	Acknowledge Interconnect Delay.
Ack	Acknowledge handshaking signal.
ADD	An instruction mnemonic which represents the DRAP cell that performs the addition operation.
ADDCOMP	An instruction mnemonic which represents the DRAP cell that performs the addition or comparison operations.
ALU	Arithmetic Logic Unit.
AMCD	Activity Monitoring Completion Detection.
ARC	Asynchronous Reconfiguration Controller, a cell in DRAP which determines when a configuration context has finished and hence controls the program counter.
ARM	Advanced Reduced instruction set computing Machine, a ubiquitous embedded microprocessor architecture.
ASIC	Application-Specific Integrated Circuit, custom non-programmable silicon created for a particular task.
ASIP	Application-Specific Instruction Set Processor, a type of microprocessors where application-specific functionality has been provided through additional high-level instructions.
CAD	Computer Aided Design.
CD	Completion Detection.
CFA	Colour Filter Array.
CGRA	Coarse-Grained Reconfigurable Array, an umbrella term for particular types of dynamically reconfigurable architectures which operate on the word level rather than the bit level.
COMP	An instruction mnemonic which represents the DRAP cell that performs the comparison operation.
CPU	Central Processing Unit, any type of processor that can perform complex control flow.
CSCD	Current Sensing Completion Detection.

CSP	Communicating Sequential Processes, a specification language for describing patterns of interaction in concurrent systems.
D_R_ID	Data Request Interconnect Delay.
DI	Delay-Insensitive, a type of delay model which categorises asynchronous circuits.
DIV	An instruction mnemonic which represents the DRAP cell that performs the division operation.
DRAP	Dynamically Reconfigurable Asynchronous Processor, the architecture proposed in this thesis.
DSP	Digital Signal Processor, a type of embedded processor with an instruction set optimised for performing common signal processing tasks.
DVB-SH	Digital Video Broadcasting – Satellite services to Handhelds transmission system standard designed to deliver video, audio and data services such as mobile television to handheld devices.
EDA	Electronic Design Automation.
FIFO	First In, First Out.
FPGA	Field Programmable Gate Array, a fine-grained reconfigurable datapath architecture mostly used in system-on-chip prototyping.
GALS	Globally Asynchronous Locally Synchronous, a design pattern used in highly multi-core architectures, to make it conceptually easier to pass information between the cores.
Gbps	Gigabits per second.
GCC	The GNU compiler collection (formerly the GNU C compiler). An open-source retargetable compiler framework.
IC	Integrated Circuit.
ID	Interconnect Delay.
IP	Intellectual property.
JUMP	An instruction mnemonic which represents the DRAP cell that controls the processor's reconfiguration.
LOGIC	An instruction mnemonic which represents the DRAP cell that performs logic operations such as XOR, AND, and OR.
ISP	Image Signal Processor/Processing, a series of algorithms that manipulate digital images, to compensate for artefacts in the sensor and optics. Can

	also refer to an ASIC implementation of this functionality.
LTE	Long Term Evolution, a 4G standard in the mobile network technology.
MDF	Machine Description File, a file format used to describe a DRAP core, in terms of cell types present, instance counts, locations in the array, timing information, and other properties.
MUL	An instruction mnemonic which represents the DRAP cell that performs multiplication operations.
MUX	An instruction mnemonic which represents the DRAP cell that performs multiplexing operations.
NRE	Non-Recurring Engineering is the initial design effort and costs spent to allow the creation of end units, irrespective of the total number of units produced.
OFDM	Orthogonal Frequency-Division Multiplexing, a radio modulation technique used in modern wireless standards such as DVB-SH and WiMAX.
PCHB	Precharge Half-Buffer circuit.
QDI	Quasi-Delay-Insensitive, a type of delay model which categorises asynchronous circuits.
	Random-Access Memory.
REG	An instruction mnemonic which represents the DRAP cell that acts as an asynchronous register.
Req	Request handshaking signal.
RGB	Red-Green-Blue pixel format.
RICA	Reconfigurable Instruction Cell Array. The dynamically reconfigurable architecture which the DRAP design is based on.
RISC	Reduced Instruction Set Computer. Also known as regular (uniform) instruction set computer, or load/store architecture.
RRC	Reconfiguration Rate Controller. A type of instruction cell in RICA which controls the program counter, affecting control flow.
RTL	Register Transfer Level.
Sbox	Switchbox.
SBUF	An instruction mnemonic which represents the DRAP interface cell that performs reading from and writing to a stream buffer.

SHIFT	An instruction mnemonic which represents the DRAP cell that performs shifting operations.
SI	Speed-Independent, a type of delay model which categorises asynchronous circuits.
SINK	An instruction mnemonic which represents the DRAP cell that writes to external logic.
SoC	System-on-Chip, a form of integrated circuit where an entire computer (CPU, memory and peripherals) is integrated into a single die or package.
SOURCE	An instruction mnemonic which represents the DRAP cell that reads from external logic.
SRAM	Static Random-Access Memory.
ST	Self-Timed, a type of delay model which categorises asynchronous circuits.
STG	Signal Transition Graphs.
VLIW	Very Large Instruction Word, a DSP processor with several operational units that are able to simultaneously execute independent instructions while sharing registers and memory.
VLSI	Very Large Scale Integration.
WiMAX	Worldwide Interoperability for Microwave Access, a 4G standard in the mobile network technology.

Chapter 1

Introduction

High-throughput mobile applications increasingly demand programmability and energy efficiency [1][2][3]. Until now, the choice of platform for such tasks has mostly been Application-Specific Integrated Circuits (ASICs), due to their best-of-breed performance and power consumption. The economies of scale possible with these high-volume markets have traditionally been able to hide the high Non-Recurring Engineering (NRE) costs required for designing and fabricating new ASICs. However, with the NREs and design time escalating with each generation of mobile applications, this practice of using ASICs is being restricted to mature and well established high-volume applications.

Designers today are looking at programmable solutions to allow them to respond rapidly to market changes and to reuse designs. This spreads costs over several generations of applications. Conventional mobile Central Processing Units (CPUs) and Very Long Instruction Word (VLIW) processors provide programmability, but cannot meet the throughput demand or performance per Watt. This forces

manufacturers to rely on custom hardware accelerators, thus sacrificing programmability, leading back to increased product lead-time and risk.

Reconfigurable datapaths offer better solutions for high throughput applications (streaming applications), when power and area considerations are also taken into account. These devices come under two main categories:

- **Fine-Grained**: Field-Programmable Gate Arrays (FPGAs) provide the ability to map any logic functions on their fine-grained (1-bit) logic and interconnect mesh. The disadvantages of using FPGAs for mobile applications are high energy consumption and area overhead, because of the large number of transistors needed to provide flexibility. Moreover, FPGAs offer reduced programmability compared to microprocessors, since developers are required to have specialised skills to convert algorithms into suitable Register Transfer Level (RTL) code for synthesis.
- **Coarse-Grained**: Most mobile application algorithms require word-wide datapaths. Coarse-grained reconfigurable computers are more suited for such applications (like multimedia and baseband) than FPGAs because they provide a massive reduction in configuration time and memory as well as complexity reduction of the placement and routing problem. There are several available coarse-grain architecture designs [3]. Despite providing improvements in high computational performance and flexibility over fine-grained architectures, they typically either do not provide enough power savings or are too difficult to program.

Recent work at the University of Edinburgh has successfully demonstrated a clocked dynamically reconfigurable datapath solution called the Reconfigurable Instruction Cell Array (RICA [4]). The processor can be programmed via conventional high-level software languages like C. This allows existing code bases and skills to be leveraged, and satisfies the programmability demand in evolving mobile applications. RICA consists of an array of coarse-grain customisable cells that can be reconfigured on every cycle.

There is a timing problem in programmable devices which must be dealt with: timing requirements change depending on the mapped datapath and the required level of pipelining. In most programmable devices, the maximum operating frequency is limited to the largest critical-path delay of all the datapaths being mapped. Through dynamic reconfiguration, RICA can improve the operating frequency of a loaded program by changing its operating frequency (via a programmable clock divider) according to which datapaths are mapped in that configuration. The critical path delay for each configuration context is estimated in software, and the resulting divider is programmed as part of the array's configuration.

Configuration contexts that loop back to themselves, also known as kernels, can have their critical path reduced via pipelining. This is done by explicitly programming registers along each routed path. RICA uses the concept of distributed registers to pipeline kernels. As a result, it requires a large clock tree to synchronise its distributed register elements. The drawback of this is poor scalability in terms of area and power: as the RICA core gets bigger, the number of sequential elements (primarily registers) in the array must be increased significantly in order to maintain the ability to pipeline. The clock tree therefore takes up a larger percentage of the area and power consumption of the core.

Next Step – Asynchronous

Asynchronous logic is a method of designing digital systems without clocks. Global synchrony is replaced with local synchronisation amongst parts that exchange data. Local synchronisation is achieved through the use of local request (req) and acknowledge (ack) signalling called handshakes [5]. The handshaking protocol implements communication and synchronisation among the components of an asynchronous datapath irrespective of its length.

1.1 Objectives and Scope of Research

This thesis presents and evaluates a novel Dynamically Reconfigurable Asynchronous Processor (DRAP) aimed at high-throughput mobile applications. By basing the architecture on RICA and using asynchronous design techniques, DRAP achieves lower power consumption than leading processors while maintaining a high level of programmability. The main distinguishing features of DRAP are as follows:

- **Event-driven energy consumption**: Asynchronous logic implements fine-grain "clock gating" by automatically turning off unused circuits since the parts of the circuit which do not contribute to the computation have no switching activity.
- **Implicit pipelining**: Most mobile baseband and multimedia programs spend over 95% of their time on configurations that loop, while the remaining time is spent in sequential configurations [4]. The asynchronous cells contain latches/flip-flops within them. As a result, a full pipelining is inherently provided for repetitive executions of datapaths. This is referred to as implicit pipelining as opposed to explicit pipelining which involves using external register cells.
- **Scalability**: When a synchronous array gets bigger, the number of sequential elements increases and hence the clock tree takes up a larger percentage of the area and power consumption of the core. By using asynchronous design techniques, the clock tree is replaced with local handshaking which implements synchronisation among the operational cells of an asynchronous datapath irrespective of its length. Hence DRAP is more scalable in terms of power and area overhead.
- **Reduced program size**: Since asynchronous cells already contain latches within them, fewer pipelining dedicated registers than equivalent clocked designs are needed. This achieves further memory savings and also reduces the area of the design since fewer switches and their configuration bits are required.

1.2 Contribution to Knowledge

The following is a list of achievements and contributions to knowledge presented in this thesis.

- Design of the DRAP system: a novel dynamically reconfigurable processor design based on RICA architecture and using asynchronous design techniques. This included designing the heterogeneous coarse-grained asynchronous cells (Section 4.4), programmable pipelined and non-pipelined interconnects (Section 1.1) and memory interfaces (Section 4.4).
- Design of the tool to generate DRAP arrays with customisable numbers and functionalities of cells and types of interconnect. This consisted of modifying the RICA toolset to take into account the asynchronous nature of the cells and different interconnect requirements. It also consisted of adding new features that allow DRAP to control its reconfiguration and also optimise its performance (Section 7.2).
- Optimised software implementations of DSP operations on DRAP (Section 8.1).
- Developed a novel clockless method of jumping between different steps of a program using handshake signals. The method allows the asynchronous reconfigurable array to retain its increased level of scalability over synchronous counterparts (Section 6.2).
- Developed a novel method for conditional acknowledge synchronisation for asynchronous interconnect design. The method allowed the design of interconnects which required simpler control and less configuration bits than other available methods (Section 5.5).

1.3 Publications

The work of this thesis is backed by the following publications:

- K. A. Fawaz, T. Arslan, and I. Lindsay, “Conditional Acknowledge Synchronisation in Asynchronous Interconnect Switch Design”, 2009 NASA/ESA Conference on Adaptive Hardware and Systems (AHS), June 2009.
- K. A. Fawaz, T. Arslan, and I. Lindsay, “Implementation of Highly Pipelined Datapaths on a Reconfigurable Asynchronous Substrate”, 2009 NASA/ESA Conference on Adaptive Hardware and Systems (AHS), June 2009.
- K. A. Fawaz, T. Arslan, S. Khawam, M. Muir, I. Nousias, I. Lindsay, A. Erdogan, "A Dynamically Reconfigurable Asynchronous Processor for Low Power Applications," 2010 Conference on Design and Architectures for Signal and Image Processing (DASIP), October 2010.
- K. A. Fawaz, T. Arslan, S. Khawam, M. Muir, I. Nousias, I. Lindsay, A. Erdogan, "A Dynamically Reconfigurable Asynchronous Processor," 2010 IEEE 8th Symposium on Application Specific Processors (SASP), June 2010.

1.4 Structure of Thesis

The thesis is divided into nine chapters. A brief outline for each chapter is described below.

Chapter 1 presents the background, objectives and scope of this research.

Chapter 2 reviews the literature relevant to this thesis. It provides an overview of the field of asynchronous circuit design and an introduction to its basic concepts and design styles. It also describes the main requirements for current and future mobile applications, explores different programmable hardware classes, and presents an overview of asynchronous reconfigurable architectures. Finally, it describes the RICA family of architectures in more detail.

Chapter 3 introduces the DRAP architecture and describes its features.

Chapter 4 describes the design of the DRAP operational cells.

Chapter 5 describes the design of the DRAP interconnect structure. It also presents a novel developed method for conditional acknowledge synchronisation for asynchronous interconnect design.

Chapter 6 describes how the DRAP architecture controls its reconfiguration. A novel method of jumping between different steps of a program for asynchronous reconfigurable architectures is presented.

Chapter 7 studies the factors that affect pipelining on an asynchronous reconfigurable architecture. It also describes the automatic tool flow for DRAP.

Chapter 8 provides an evaluation of the DRAP architecture and compares it with RICA and other leading technologies.

Chapter 9 summarises the general conclusions of the whole thesis and makes recommendations for further work.

Chapter 2

Literature Review

This chapter begins by providing an overview of the field of asynchronous circuit design and an introduction to its basic concepts and design styles. It then describes the main requirements for current and future mobile applications followed by an exploration of different programmable hardware classes in general and asynchronous reconfigurable architectures in particular. Finally, the chapter describes the RICA design in more detail.

2.1 Asynchronous Circuit Design

Due to the growth in the number of transistors on Integrated Circuits (ICs), Very Large Scale Integration (VLSI) systems are increasing in complexity and size. As a result, large parameter variations across a chip are making it exceedingly difficult and expensive to control delays in global signals such as clocks. This, along with issues of power management and modularity, has led the semiconductor industry to give serious consideration to the adoption of asynchronous circuit technology [6].

The last two decades have seen a revival in research into asynchronous circuits [6][7][8][9]. Recent companies that provide asynchronous tools and products include: Achronix Semiconductor [10], TIEMPO [11], and Fulcrum Microsystems [12].

This section aims to provide an overview of the field of asynchronous circuit design and an introduction to its basic concepts and design styles. Given the size of the field, it is impossible to cover all the available design methods in detail. However, there are numerous citations that provide the alert reader with the direction for an in-depth study of any particular method.

2.1.1 Definition and Basic Concepts

Circuit design styles can be classified according to two major categories, synchronous and asynchronous. Synchronous circuits are simply defined as circuits which are sequenced by one or more global timing signals known as clocks. Asynchronous circuits cover a range of different circuit styles which reject lock-step synchronous operation by rejecting the use of global periodic clock signals [13].

The added value of asynchronous circuit design can be best understood by reviewing key properties of synchronous circuits. Figure 2.1 shows a simple block diagram of synchronous circuits.

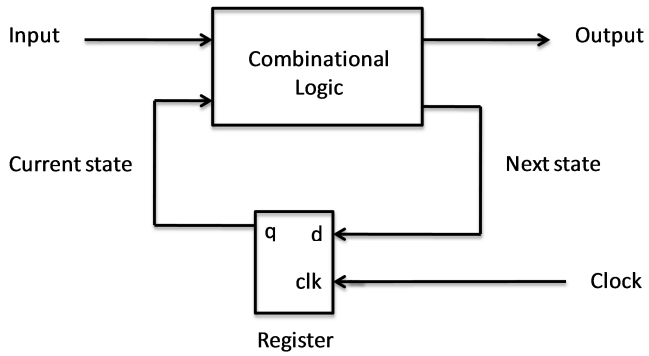


Figure 2.1: Synchronous circuit block diagram [14].

The following applies [14]:

1. The fixed clock period is determined by the longest delay path through the combinational logic.
2. The register dissipates energy during each clock cycle regardless of whether a change in state has occurred.
3. The clock signal modulates the overall supply current, causing peaks in power-supply noise and hence electro-magnetic emissions to occur.
4. All functional sub-modules operate in lock-step. This requirement is at odds with the growing significance of interconnect delays and the heterogeneous nature of System-on-Chip (SoC) architectures.

The shortcomings of synchronous circuits provide the motivation and potential advantages of asynchronous design. These are summarised below [15][16]:

- Average/Best-case Performance – an asynchronous system operating speed is determined by actual local latencies unlike a synchronous system which adopts worst-case performance to determine its global clock period.
- Low Power Consumption – each stage is controlled by demand and hence idle stages do no work and consume no dynamic power. This can lead to lower power designs and zero standby dynamic power consumption.
- Modularity – asynchronous circuits provide inherent modularity because their interfaces are free from global constraints.
- Lower Electro-magnetic Noise – different stages in an asynchronous circuit operate at different speeds and switching activity is both operation and data-dependent. Therefore, electro-magnetic noise is more evenly spread over the operation time and not fixed to a particular frequency.
- Robustness towards Variations in Supply Voltage, Temperature, and Fabrication Process Parameters – the timing in asynchronous circuits can be insensitive to circuit and wire delays or based on matched delays.

- No Clock Distribution/Clock Skew Problems – asynchronous circuits have no global signal that needs to be distributed with minimal phase skew across the circuit.

There are obstacles that need to be overcome for asynchronous design techniques to challenge synchronous ones in use in VLSI systems. All asynchronous circuits have an additional operational constraint in dealing with glitches when compared to synchronous counterparts.

A *glitch* is a result of a non-monotonic transition between one voltage level and the other; a circuit containing an output which may glitch is said to contain a *hazard* [13]. In a synchronous circuit, a glitch on a clock signal will typically cause the circuit to malfunction whereas glitches on non-clock signals do not cause a malfunction as long as the signal becomes stable for a certain time before and after a clock signal transition. This means that non-clock signals in synchronous circuits do not need to be designed hazard free; this results in smaller circuits. The clock signal however must be carefully controlled and distributed which proves to be costly in terms of area.

Asynchronous circuits on the other hand may require more gates than a functionally equivalent synchronous circuit in order to achieve hazard-free operation. Additionally, there are several techniques in synchronous design to overcome the disadvantages of worst-case timing and high power consumption and clock skew. These include using finely-grained pipeline structures, clock distribution and de-skewing methods and using clock switches to disable the clock in parts of the design when not needed (power gating) [13]. Discussing these techniques is beyond the scope of this thesis; however, it is worth noting that clock management is a difficult problem and all proposed solutions to this problem incur high costs.

Disadvantages of asynchronous design are summarised as follows [13][15][16]:

- Increased Circuit Cost – extra circuitry is required to allow for locally timed operation and to eliminate hazards. This circuitry is required extensively throughout the design and can add heavy penalty in area, speed and power consumption.
- Complexity – asynchronous circuit design deals with problems of data-validity and hazards explicitly by either using a complex design style or inserting delays which require exhaustive simulation to validate.
- Shortage of tools – there are a number of different Computer Aided Design (CAD) tools that can aid the construction of asynchronous circuits, as shown in [17]; however, most have been developed within research groups and are far from commercial products.
- Testability – because of lack of global timing reference, synchronous test techniques such as Scan-paths are difficult in asynchronous designs. Additionally, testing is complicated by the special design constraints of asynchronous circuits. For example, redundant logic is used in asynchronous circuits to eliminate hazards and this makes testing more difficult.

2.1.2 Handshaking

Handshaking is a signalling scheme used by most asynchronous circuits. A handshake involves the use of two basic control signals: a *request* signal, sent from an *initiator* to a target, which initiates an action and a corresponding *acknowledge* signal, from the target to the initiator, which signals the completion of that particular action (Figure 2.2). These handshake signals are independent of any global timing system. They are only concerned with the local temporal relationships between two systems sharing an interface [13].

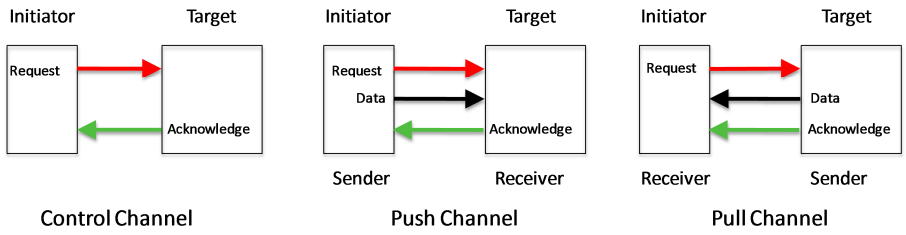


Figure 2.2: Handshaking: control and data (push and pull) channels [16].

The medium upon which two sub-systems communicate by handshaking is called a *channel*. This includes the control channel (request and acknowledge signals) and the data channel (Figure 2.2). Furthermore, there are two types of data channel. These are determined by the direction of the data flow [16]:

- Push Channel – data flows from initiator to target where the request signals validity of arriving data and the corresponding acknowledge signals the successful receipt of data.
- Pull Channel – data flows from target to initiator where the request asks for data to be sent and the corresponding acknowledge signals its arrival.

2.1.3 Signalling Protocols

There exist several choices of how to encode the alternating events of the request and acknowledge onto specific control wires. The two most pervasive signalling protocols are described below [13][15][16]:

- Two-Phase signalling – also known as transition signalling or non-return-to zero signalling (Figure 2.3). In this protocol, each transition has a meaning, i.e. each transition on a wire signals a request or acknowledge. This means that two transitions are needed to complete a handshaking event.

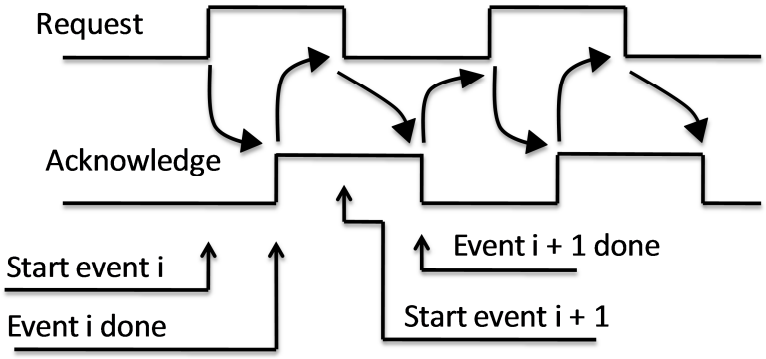


Figure 2.3: Two-phase asynchronous signalling protocol [13].

- Four-Phase signalling – also known as level signalling or return-to zero signalling (Figure 2.4). This protocol uses the logical level of the request and acknowledge wires to control the handshake. To achieve this, both wires must return to logic zero at the end of a handshake. This means that four transitions are needed to complete an event.

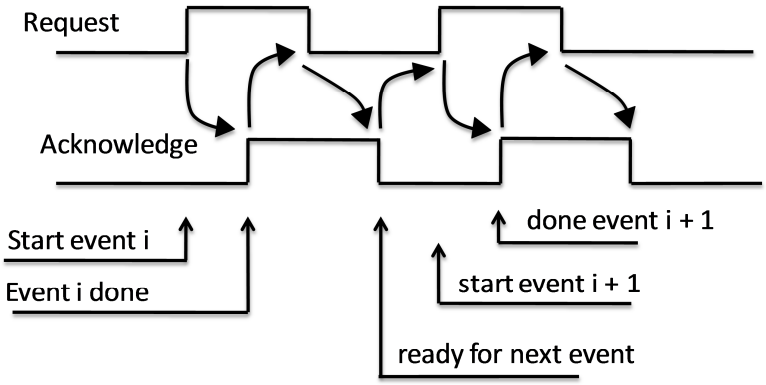


Figure 2.4: Four-phase asynchronous signalling protocol showing an early scheme [13].

Additionally, because each handshake is made up of four phases, there are several choices which can be made about the validity of the data within handshakes depending on whether the request/acknowledge is chosen to be the positive or negative edge of the corresponding control wire.

This translates to three main data validity schemes for four-phase signalling: early (request and acknowledge both positive edges), late (request and acknowledge both negative edges), and broad (request positive edge and acknowledge negative edge).

Four-phase signalling leads to simpler and smaller hardware than two-phase signalling since it is sensitive to only one edge. Additionally, it allows more flexibility when transferring data (early, broad, late modes). Two-phase signalling has the potential to be faster and more energy efficient than four-phase signalling because it uses each transition in a handshake. However, this is not necessary the case, as two-phase hardware implementations may require more logic complexity than equivalent four-phase ones [18]. This increased logic complexity consumes more power than is saved by the reduced control transitions.

This was shown to be the case in the two versions of the asynchronous ARM processor produced by the University of Manchester. ARM2 [18], designed using four-phase signalling, demonstrated a performance and low-power improvement over ARM1 [19], designed using two-phase signalling.

For more than two module interfaces, protocols based on similar sequencing rules exist [13]. These require the conjunction of two or more request or acknowledge signals to provide a single corresponding request or acknowledge. The element used for this purpose is the Muller C-element (Section 2.1.4), which is one of the most common components within any asynchronous design. The C-element effectively merges two requests into a single one and hence allows three subsystems to communicate using a two or four phase signalling.

2.1.4 Muller C-element

This component was first defined by Muller [20] as a way of synchronising two events. The C-element is a state-holding element, much like the synchronous flip-flop. Figure 2.5 shows the common logic symbol and a gate-level implementation of

the C-element. It is described by the following production rules: if the inputs are equal, then make the output equal to the input; if the inputs are different, the output holds its current value.

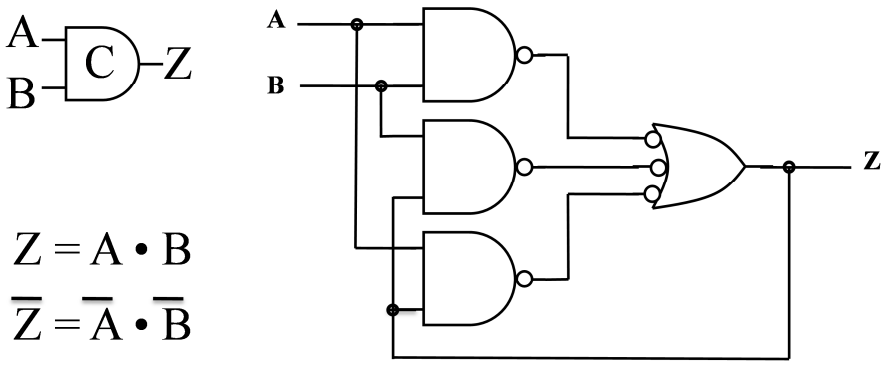


Figure 2.5: Symbol (left) and gate-level implementation of a Muller C-element (right).

2.1.5 Data Encoding

Sections 2.1.2– 2.1.4 describe and present several choices for control channels. There are also multiple options for how data is encoded in the data channel of an asynchronous circuit. These options are independent of the signalling protocol choices:

- Bundled Data – also known as single rail encoding. One wire for each data bit and a separate wire to indicate validity of the entire data (usually the request or acknowledge wire) (Figure 2.6). The datapath in this case is much like a synchronous one. This approach uses fewer wires than other data encoding techniques for asynchronous design; however, it relies on the timing assumption that the data is valid before the data valid signal is raised.

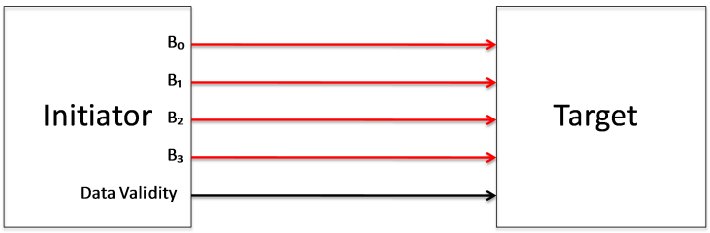


Figure 2.6: Bundled data encoding - each bit is represented by one wire; one wire (request/acknowledge) to indicate validity.

- Dual-rail encoding – also known as 1-of-2 encoding. Two wires for each data bit (Figure 2.7), one wire to indicate the bit is high and one wire to indicate it is low. A typical dual-rail encoding has four states:
  - Idle (data not valid) – 00.
  - Valid Low – 10.
  - Valid High – 01.
  - Illegal – 11.

  The receiver must check for the validity of all n-bits before using them or asserting the acknowledge signal (see Section 2.1.6). This method is delay-insensitive. However, it leads to increased complexity in both wiring and logic when compared to bundled data.

Figure 2.7: Dual-rail encoding - each bit is represented by two wires; the target in this case has to check for the validity of the received data bits before using them or asserting the acknowledge signal.

- N-of-M codes – this is a generalisation of the 1-of-2 encoding, where N of the M wires are ever active. The form 1-of-4 also results in two wires per bit and potentially leads to lower power consumption as fewer transitions on the wires occur as compared to dual-rail. However, such codes are non systematic for N greater than 2 and hence pay a high cost in logic complexity [15].

Bundled data and dual-rail schemes are the most commonly used in asynchronous circuits today. Other encoding schemes have been proposed; these are summarised in [21].

2.1.6 Completion Detection

Completion Detection (CD) is an important aspect of asynchronous circuits. In a signalling protocol, a completion signal must be generated in order to control the acknowledge signal. Some methods are discussed below:

- For dual-rail encoding, the acknowledge signal can be generated by using the exclusive-OR of the outputs. This technique works directly with four-phase signalling. With two-phase signalling, extra logic is required. As the number of outputs increases, the design of the CD circuit becomes slower and hence reduces the overall speed of the asynchronous circuit. Designing fast CD circuits is key for designing high speed asynchronous circuits. For more information, the reader is pointed to the papers [22][23][24] which contain a performance evaluation of several CD circuits.
- For bundled data encoding, conventional synchronous timing analysis of the datapath is used to determine the time taken by the circuit to compute a valid result after a request has been received. A corresponding delay element made up of inverters or other gates is produced. This delay element takes in the request signal as input, delays it for a time greater than that of the worst case and turns it into an acknowledge signal. This technique works well with both four-phase and two-phase signalling. The disadvantage of this method is that

it does not exploit data dependency and as a result, leads to worst case performance.

- Other techniques which work with single-rail encoding schemes have been developed. These include Current Sensing CD (CSCD) [25][26][27][28] which uses the low current characteristics of quiescent CMOS circuits to detect when transition has ended on a datapath. Another is Activity Monitoring CD (AMCD) [29] where the activity of internal modes is monitored to determine whether the circuit is still switching.

2.1.7 Operation Modes

This section describes the different modes of interaction between the circuit being designed and its environment. Asynchronous circuits operate using one of the following modes or their derivatives [15][16]:

Fundamental mode – this design mode was devised by David Huffman in the 1950's [30]. The circuit is assumed to be in a state where all signals (input, internal, and output) are stable. After changing one input signal, the environment must wait for the circuit to stabilise before any more inputs can be changed. Since the environment does not know internal signals, the longest delay in the circuit must be calculated and the input signals must be kept stable for at least this delay amount.

Burst mode – this is a generalised form of the fundamental mode, designed in the late 1970's [15], where a restricted form of multiple-input and multiple output changes is allowed. When in a stable state, a burst mode circuit reacts by computing a burst of output changes. The environment has to wait for the circuit to stabilise before applying another input burst. There exist several mature tools for synthesising burst mode controllers; the most sophisticated is Minimalist, which has been developed in academia at Columbia University [31].

Input-Output mode – this design mode was pioneered by David Muller in the 1950's [32]. The circuit is assumed to be in a stable state, each input change results in a

corresponding output change. There are no assumptions about internal signals and the environment may change the inputs before the circuit has stabilised in response to previous input changes. Because of the causal relationship between the input and output transitions in this mode, the interfaces may become complex [16]. For this reason, trace based methods such as *signal transition graphs* [33] and *Petri-nets* [33] are used to specify and model these circuits.

2.1.8 Delay Models

In synchronous circuit design, clocking allows designers to ignore timing, switching order, glitches and races; circuits based on such design can be regarded as instantaneous operators, which compute a new result at each clock cycle [13]. On the other hand, asynchronous circuit designers need to take into account all of the above since any of these problems may lead to incorrect circuit behaviour; asynchronous circuits can be regarded as computing dynamically through time. Therefore, a delay model is critical in defining the dynamic behaviour of such circuits.

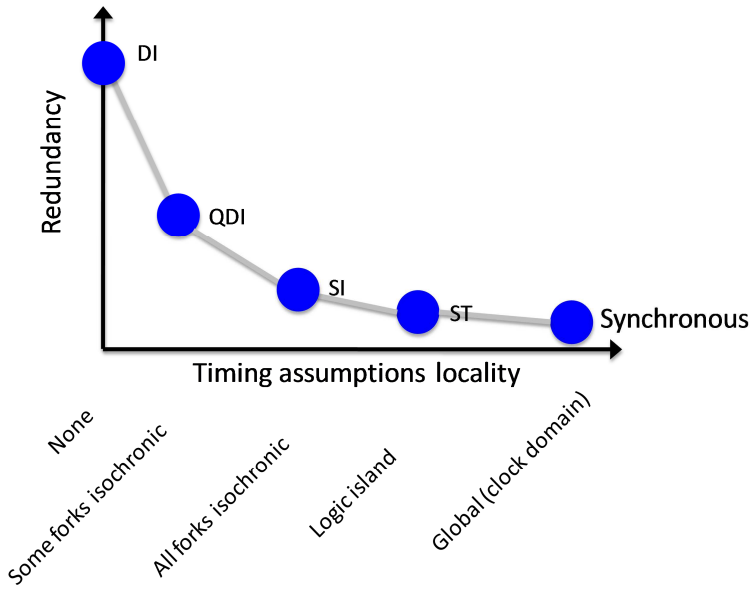


Figure 2.8: Comparison of delay models - redundancy required to eliminate hazards versus locality of timing assumptions [34].

Delay models categorise circuits by the propagation delay assumptions of the circuit components. Such models provide a designer with a template for construction and verification of the circuit.

The two main types of delay models are:

- Unbounded delay – a delay may have any finite value.
- Bounded delay – a delay may have any value within a given range.

Given these models, asynchronous circuits can be classified at the gate-level as being one of the following [13][15][16][34]. Figure 2.8 illustrates the redundancy versus locality of timing assumptions for each of the delay models:

- Delay-Insensitive (DI) – a DI circuit is one that operates correctly assuming an unbounded gate and wire delay model. Unfortunately, the class of DI circuits is very limited; it has been proven by Martin [35] that only circuits built using C-elements and inverters can be DI.
- Quasi-Delay-Insensitive (QDI) – a QDI circuit is a circuit that is DI with the exception of carefully identified wire forks, where the skew between different branches of a fork is assumed to be smaller than the minimum gate delay. These forks are known as *isochronic forks*. QDI circuits are the least compromise to DI circuits which allows the design of practical circuits using simple gates and operators [13].
- Speed-Independent (SI) – David Muller introduced this class of circuits in the 1950's [32]. An SI circuit is one that operates correctly assuming unbounded gate delays and ideal wire delays (zero delay). However, the latter assumption is becoming increasingly impractical for large designs and with shrinking feature size where the share of wire delays is growing. Input-output mode circuits (Section 2.1.7) are usually referred to as SI circuits.
- Self-Timed (ST) – a self-timed circuit is one whose operation relies on more elaborate engineering timing assumptions to ensure correct operation. An

example of self-timed circuits is Matched Delay circuits, which use delay lines to match the delay of a combinational logic island.

2.1.9 Muller and Huffman Models

When designing asynchronous control circuits, a combination of the following design choices must be taken into account: specification formalisms, gate and wire delay models, and operation modes. These combinations have led to a multitude of approaches and theories being proposed for asynchronous control circuits.

Historically, two main models of operations for asynchronous circuits have been used [15][36][37]:

- Huffman model: the circuit is decomposed into a combinational logic block and feedback signals with delay elements. The bounded delay model (Section 2.1.8) is associated with all interconnections. The circuit initially functioned under the fundamental mode and was later generalised to work under burst mode (Section 2.1.7).
- Muller model: the circuit is decomposed into gates with arbitrary interconnections and feedback signals made of wires. It uses the unbounded delay model (Section 2.1.8) for gates and functions under the input-output mode (Section 2.1.7).

The synthesis techniques for Huffman models are based on *Finite state machines.* A flow table and special state assignment algorithms are used to avoid the operation of the circuit depending on the delay of feedback wires. For Muller models, the total system including the environment must be modelled as a state transition diagram; this is used for both analysis and synthesis of the circuit. [36]

Muller circuits are more robust, portable, and easier to verify. They are more closely related to modern asynchronous circuits. However, they may be larger and slower

than Huffman circuits. Huffman circuits on the other hand have robust circuitry but impose strict timing restrictions on the environment and feedback paths. [37]

2.1.10 Specification and Synthesis of Control Circuits

This section describes briefly how burst mode circuits and SI circuits (input-output mode circuits) are synthesised. The section does not go into details of how available tools work nor does it describe the evolution of the specification and synthesis techniques. For more detailed information, the reader is pointed to chapter 6 of [15] and section 6 of [13] both of which contain a more comprehensive study of the subject as well as pointers to relevant literature. The reader should refer to [17] for a list of asynchronous synthesis and test tools.

2.1.10.1 Burst Mode Circuits

The asynchronous controller here is viewed as a finite state machine and its specifications are described using a flow table or state table. Burst mode specifications were introduced by Davis [38] to allow more concurrency than fundamental mode state machines. Burst mode circuits have robust combinational circuitry. The circuits are guaranteed hazard-free under all possible gate and wire delays in the environment [39].

Many CAD tools and algorithms have been developed for the synthesis and verification of burst mode circuits. These include the locally clocked method [40], three-dimensional (3D) method [41], MEAT [42], and MINIMALIST [43]. Each of the methods performs the following steps [39]:

- State minimisation and assignments using constraints to avoid hazards and races.
- Hazard free logic minimisation.
- Technology mapping.

2.1.10.2 Speed Independent Circuits

The asynchronous system here is viewed as a partially-ordered sequence of events and not as state-based [13]. *Petri nets* [33], such as *Signal Transition Graphs* (STGs) or I-Nets, are used to specify such circuits. Petri nets can be used as a structural specification or a behavioural specification. The latter is more common in modern synthesis methods, where a Petri net is transformed into a *state graph*, describing the explicit sequencing behaviour of the net. Some researchers use state graphs for specification as an alternative to Petri nets. Many tools such as SIS [44] and most popularly PETRIFY [45][46] have been developed for STG synthesis. The general procedure followed by such tools is [15][39]:

1. Capture the behaviour of the circuit and its environment in an STG.
2. Generate corresponding state graph; add state variables if needed.
3. Derive Boolean equations for outputs and next state functions.
4. Decompose high fan-in gates so as to preserve speed independence and map onto a library of gates.

There are other important techniques for specification and synthesis of SI circuits. These methods are known as transformation methods; the asynchronous system in this case is viewed as a collection of communicating processes. A system is specified as a program in a high-level language. Almost all available languages that are used in modelling and synthesis of asynchronous circuits belong to the Communicating Sequential Processes (CSP) family of languages. The program is transferred by a series of steps, into a low-level program that maps directly into a circuit. The transformation is done using algebraic or compiler techniques to carry out the translation. [13][15]

There are many tool based compiler transformation methods. These include CAST [17] based on Martin's translation process [47], TiDE (Handshake Solutions) [48], developed at Philips and based on a new CSP language called Tangram [49] (later

renamed Haste) and Balsa [15][50] developed at Manchester University and based on a new CSP language called Balsa, which is based on Tangram.

2.1.11 Asynchronous Datapaths

There exist several techniques and structures for designing synchronous and asynchronous datapaths and controllers. Modern datapath design is often done using pipelines. In synchronous pipelines, data advances at a fixed clock rate. The clock cycle must be set to the slowest stage and clock skew and stage latency must be taken into account. This results in a synchronous pipeline operating far slower than its potential performance. Additionally, changing the depth of a synchronous pipeline requires extra effort to change the clock frequency and to make sure the behaviour of the system is unchanged.

An asynchronous pipeline is not globally clocked. Each stage may pass data to its neighbour whenever it is done and the next stage is free. Different stages may operate at different speeds and complete early depending on the data. This results in elastic pipelines where any pipeline stage can vary its processing time without any timing restrictions [51]. The depth of an asynchronous datapath can be varied without changing the behaviour of the system since handshaking protocols implement communication and synchronisation among the components of the datapath irrespective of its length. Manohar and Martin showed in [52] that all asynchronous systems that do not exhibit arbitration have elastic pipelines.

As for synthesising asynchronous datapaths, some of the tools described in Section 2.1.10 are also appropriate for datapath synthesis. These are the CSP based tools:

- CAST: synthesises QDI 4 phase dual-rail circuits.
- Handshake solutions/Balsa: synthesises 4 phase bundled data circuits.

2.1.12 Testing and Synthesis for Testability

Testing is needed to validate the correctness of any fabricated circuit. In the production of synchronous chips, testing and synthesis for testability play a vital part. In asynchronous circuits, testing is complicated by the large variety of asynchronous design approaches and constraints. As an example, redundant logic, which is used by asynchronous circuits to eliminate hazards, also makes testing more difficult. There is a great deal of ongoing activity in the field of testing asynchronous circuit. The discussion of the topic is beyond the scope of this thesis. For more information on the subject, [13], [53], and [54] provide a good starting point.

2.2 Requirements for Mobile Applications

Mobile systems are being called upon to run multimedia applications traditionally associated with desktop computers. Current high-end mobile devices integrate high-bandwidth internet access, high-definition video processing, interactive video conferencing and voice telephone into small packages. Future handheld devices will also offer many new functionalities and services such as complex image processing (multimedia applications) and support for new wireless standards such as WiMAX [55] and Long Term Evolution (LTE) (baseband applications) [56][57]. [58][59][60]

An idea of the challenges awaiting next generation mobile devices can be seen in the fourth-generation (4G) wireless technology proposed by the International Telecommunications Union [61]. They propose that 4G technology increase bandwidth to maximum data rates of 100 Mbps for high mobility such as mobile access and 1 Gbps for low mobility such as local wireless access [62]. This is equivalent to a ten to 1000 times increase in computational requirements over current third generation (3G) wireless technologies, with a power budget of approximately 1 Watt for all the computation. Other forms of signal processing, such as high-definition video, are also up to 100 times more computationally intensive than current mobile video. [63]

This section discusses the main requirements of processors aimed at running current and future mobile applications.

2.2.1 High Performance and Energy Efficiency

The high performance requirement for mobile devices is caused by two important factors:

- Supporting multimedia applications: these include current and future applications such as high-definition video processing, interactive video conferencing, audio and video multi-way communication and other entertainment applications.
- Supporting wide range of wireless communication standards: current 3G and future 4G standards and their worldwide variations.

Even though processors must deliver high performance to deal with the above mobile applications, optimising processors only for processing speed will lead to systems that are extremely power hungry. Mobile devices are typically battery powered, have a limited amount of available energy, and are restricted in size and weight. As a result, the mobile system should perform more work with the same or even smaller amount of energy. This means mobile devices have to also be energy-efficient.

2.2.2 Flexibility and Programmability

Wireless communication standards are continuously evolving and there are multiple variations within each generation of mobile technology. Current and future mobile devices are expected to be worldwide and support these different standards and their variations. The evolution of existing standards also applies to multimedia applications where standards are changing over time or replaced by new ones. Until recently, the economies of scale possible with high-volume markets have been able to hide the high NRE costs required for designing and fabricating new devices to

target the evolving standards. However, with the NREs and design time escalating with each generation of mobile applications, this practice is reaching its limit.

Hardware used in future mobile devices should not become obsolete each time an existing wireless or multimedia standard is changed or a new one is introduced. This requirement of mobile devices is defined in this thesis as flexibility. Designers today are looking at programmable solutions to achieve this flexibility. Programmable solutions allow them to respond more rapidly to changes in the market and to reuse designs and hence spread costs over several generations of applications.

The final requirement of mobile devices is that they should be highly programmable. A highly programmable device is defined in this thesis as one that can be easily programmed without the need for specialised programming skills.

2.3 Reconfigurable Computers and Dynamic Reconfigurability

As concluded from the discussion in the previous section, an architecture for future mobile applications must provide high performance and energy efficiency. It must also be flexible and easy to program. ASIC designs suffer from inflexibility and very high NRE costs. They cannot fulfil the requirements of future mobile applications. As a result, designers are turning towards programmable devices to find a solution that can address these requirements. Programmable devices allow designers to respond more rapidly to changes in the market, to reuse designs and to spread costs over several generations of applications.

In this section, different programmable hardware classes that perform tradeoffs between the different and somewhat contradicting mobile application requirements are explored. The benefits and drawbacks of applying asynchronous design techniques on such architectures are presented.

2.3.1 Microprocessors

Microprocessors such as conventional mobile CPUs and DSPs provide the highest levels of programmability among programmable hardware. However, they cannot meet the throughput demand or performance per Watt required for next generation mobile applications. This forces manufacturers to rely on custom hardware accelerators. This in turn sacrifices programmability, leading back to increased product lead-time and risk.

VLIW processors rely on compiler optimisation to achieve computational efficiency. However, they only offer high throughput for algorithms that have a high level of instruction-level parallelism. Many algorithms do not have the latter, in which case they perform slower than microprocessors, whilst still consuming more power. The NRE for VLIWs is also higher, because the tools often require a lot of hand optimisation to keep the Arithmetic Logic Units (ALUs) full.

Microprocessors are small and simple due to their tightly integrated structure with a few ALUs. There have been several attempts to design asynchronous microprocessors [64][65][66]. Even though microprocessors have small clock trees relative to the total area, applying asynchronous techniques to them and eliminating the clock tree can still prove beneficial. This is the case of the asynchronous ARM processor, the ARM996HS [67]. It was designed using the TiDE tool as a collaboration between ARM and Handshake Solutions. Compared to an equivalent synchronous ARM processor, the ARM996HS consumes 2.8x less power and reduces current peaks by a factor of 2.4 leading to lower electromagnetic emissions. This came at a cost of 23% reduction in maximum achievable frequency and 10% area increase [67].

2.3.2 Reconfigurable Computers

Figure 2.9 shows a plot of the three main families of devices along an axis of programmability/performance. ASICs and Microprocessors fail to address the

requirements of future mobile applications. This has resulted in designers looking towards reconfigurable computers, the class of devices that lies between ASICs and microprocessors [3].

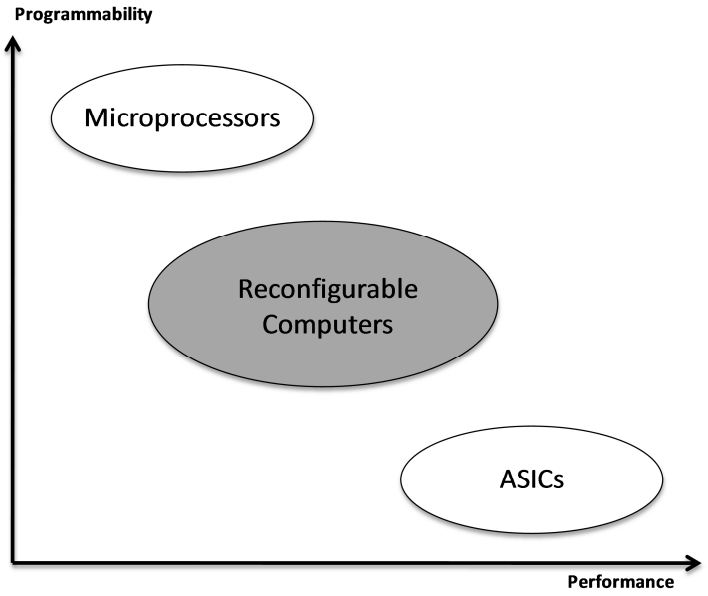


Figure 2.9: Position of reconfigurable computing (adapted from [68]).

Reconfigurable computers are defined as programmable fabrics where a circuit/datapath is mapped for execution. Reconfigurable datapaths offer better solutions for high throughput applications when power and area considerations are also taken into account. Reconfigurable hardware could be the key to high-throughput, energy-efficient, yet flexible architectures.

There are several ways of classifying reconfigurable computers. One way is according to their granularity: they could be based on fine-grained or coarse-grained functional units. For a more detailed description of reconfigurable computers and their different classifications, the reader is referred to [3].

2.3.2.1 Fine-Grained Arrays

FPGAs, such as those provided by Altera [69], Xilinx [70], and Actel [71], are fine-grained reconfigurable fabrics whose interconnect and operations are at the bit-level. The fine-grain aspect of FPGAs makes them extremely flexible and suitable for a very wide range of applications. They have the ability to map any function on their fine-grained operation and interconnect mesh. The large number of transistors and switching needed to provide flexibility incur a tremendous area overhead, large configuration context, and high energy consumption. Moreover, FPGAs offer reduced programmability compared to microprocessors, since developers are required to have specialised skills to convert algorithms into suitable Register Transfer Level (RTL) code for synthesis. This prohibits the deployment of fine-grained FPGAs in mobile applications.

The sea of fine-grained operational units that make up an FPGA is difficult to synchronise and requires a large power hungry clock tree structure. As a result, FPGAs can benefit from the application of asynchronous techniques in order to eliminate the large clock tree and introduce average-case performance and micropipelining. Section 2.4.1 provides examples of asynchronous FPGA designs.

2.3.2.2 Coarse-Grained Reconfigurable Arrays

Whereas fine-grained arrays are more suitable for applications involving bit or irregular sized data manipulations, Coarse-Grained Reconfigurable Arrays (CGRAs) are more suited for the growing multimedia and streaming applications where the majority of operations are performed at the word level.

CGRAs share similar interconnect concepts as FPGAs, but require fewer functional units to implement a given task. The coarse granularity and reduction in functional unit count lead to a reduction in area overhead. They also provide a considerable reduction in the complexity of the placement and routing problem. They further lead

to a massive reduction in configuration time and memory as well as to a potential reduction in the total energy consumed per computation.

Having fewer functional units than FPGAs makes them an even better candidate for applying asynchronous techniques to them. CGRAs also have a large clock tree because of the large number of functional units. As a result, they too may benefit from eliminating the clock tree and introducing average-case performance. Applying asynchronous techniques requires extra logic at each functional unit to perform local synchronisation. The extra logic translates to an increase in area of the functional unit. Because the CGRAs' functional units are larger in size than those of FPGAs, their relative area increase due to the extra logic (asynchronous control) is much smaller.

For the work presented in this thesis, the synchronous coarse-grained RICA was chosen as the base architecture for designing a new processor that targets high-throughput mobile applications. Section 2.1 of [72], Section 2.1 of [73], and Chapter 2 of [74] provide a comprehensive study of the different classes of reconfigurable computers and give examples for each. From those studies, it can be seen that the RICA architecture is unique in providing a high level of programmability, energy efficiency and high performance suitable for demands of high-throughput mobile applications such as multimedia. Additionally, the coarse-grained nature of the RICA architecture along with its large clock tree make it a great candidate for applying asynchronous design techniques on it in order to improve energy efficiency and scalability. Finally, the author of this thesis had full access to the RICA designers and tools. This facilitated reusing parts of the RICA designs in DRAP.

2.4 Asynchronous Reconfigurable Architectures

Several proposals for asynchronous reconfigurable architectures have appeared in literature. The trend in designing such architectures has been to pick an existing synchronous design as a starting point and then to apply asynchronous design

techniques to it. In general, asynchronous logic has been used in reconfigurable synchronous designs to either improve throughput or to reduce power consumption. Table 2.1 contains the categorisation of some asynchronous reconfigurable computers.

Table 2.1: Categorisation of asynchronous reconfigurable computers.

Design	Encoding	Signalling	Timing Constraints	Granularity
Hauck (1994) [75]	Dual-rail	4 phase	QDI	Fine
Maheswaran (1995) [76]	Bundled	2 phase	Matched delay	Fine
Payne (1996) [77]	Bundled	4 phase	Matched delay	Fine
Manohar (2004) [86]	Dual-rail	4 phase	QDI	Fine
Achronix [10]	Dual-rail	4 phase	QDI	Fine
Martin (2003) [87]	Dual-rail	4 phase	QDI	Fine
Sun (2006) [90]	Dual-rail	4 phase	QDI	Coarse
Mishra (2006) [91]	Bundled	4 phase	Matched delay	Coarse
DRAP [1][2]	Bundled	4 phase	Matched delay	Coarse

An intrinsic characteristic of FPGAs is the presence of a very large clock tree that is needed to synchronise the sea of fine-grained operational units. Applying asynchronous techniques to the design of FPGAs and hence eliminating the large clock tree and introducing average-case performance could provide great benefits such as reduced power consumption and increased throughput. As a result, most of the asynchronous reconfigurable attempts targeted the family of FPGA architectures. Section 2.4.1 gives an overview of work done in this field.

More recently, there have been a few proposals for coarse-grained reconfigurable computers whose basic cells compute functions of up to 64 inputs. This follows the general trend of evolution in reconfigurable devices from fine-grained FPGAs towards coarse-grained programmable cores, influenced by programmability aspects in order to increase usability and reduce power [3]. Coarse-grained programmable architectures also require a large clock tree to synchronise their distributed

operational cells. However, because of the coarse-grained nature of their cells, the area overhead of adding asynchronous control logic is much lower than for FPGAs. Section 2.4.2 gives an overview of the available asynchronous coarse-grained architectures.

2.4.1 Asynchronous FPGAs

Early designs [75][76][77] were based on modifying existing synchronous FPGA architectures. MONTAGE [75] was the first reconfigurable asynchronous logic, an FPGA that was capable of implementing both synchronous and asynchronous circuits. It was developed at the University of Washington and is based on a synchronous FPGA called TRIPTYCH [75] that was also developed at the university. TRIPTYCH is extended by the addition of specific arbiter blocks and modifying the functional units. PGA-STC [76] is another asynchronous FPGA. It is similar to MONTAGE but with the addition of a reconfigurable delay line targeted at the implementation of two-phase bundled data protocol. Another asynchronous FPGA architecture, STACC [77], based on fine-grain FPGA architectures, is dedicated to the implementation of four-phase bundled data systems. The clock is replaced by control signals generated by an array of timing cells.

There are some approaches that use synchronous FPGAs to implement bundled data encoded asynchronous designs [78][79][80]. The purpose of these designs is to use existing synchronous FPGAs to prototype asynchronous logic. In [81] a synchronous FPGA that is targeted to several different asynchronous design styles is presented. Other methods map clocked netlists onto asynchronous logic blocks: [82] maps the FPGA logic blocks onto two phase dual-rail micropipelines while [83] uses phased logic gates to implement the FPGA LUT design. The resulting asynchronous FPGA designs use non-pipelined interconnects.

Another approach to reconfigurable asynchronous devices applies Globally Asynchronous Locally Synchronous (GALS) design techniques to conventional synchronous FPGAs by partitioning it into smaller blocks of FPGA cells. The local

connections within a block are synchronous to a local clock and the blocks connect to each other via four-phase bundled data protocol [84][85].

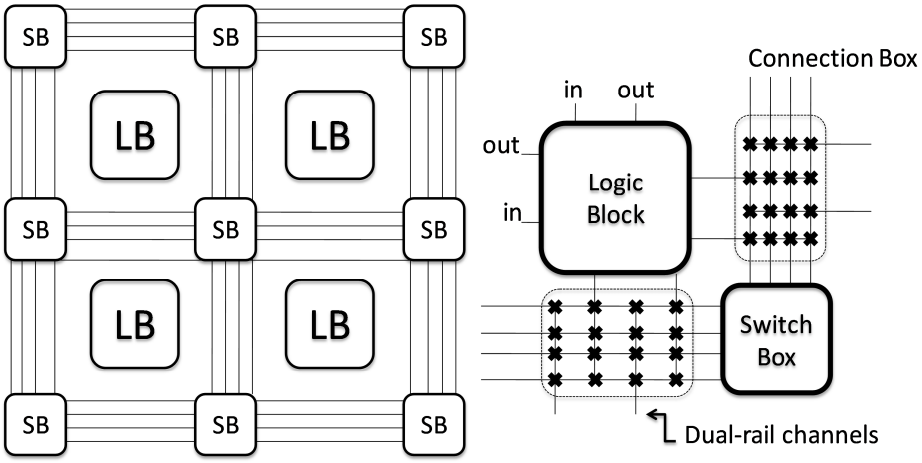


Figure 2.10: Overview of the topology used in the asynchronous FPGA of [86].

More recently there have been attempts to design fully asynchronous reconfigurable architectures that can achieve high throughputs. In [86][87], asynchronous dataflow-based fine-grain FPGAs using finely pipelined dual-rail asynchronous logic cells with four-phase handshaking are presented. The two designs use the island-style interconnect topology found in typical FPGAs [88] with connection boxes and pipelined switch boxes (Figure 2.10). From among various QDI circuit templates [89], both designs use the Precharge Half-Buffer circuit (PCHB) family, which provides compact designs with high throughput and low latency. However, the choice of cells and the cluster architecture between both designs are different. In [87], the number of active channels and the conditions of communication of the logic cells can be configured, hence achieving programmable communication patterns. In contrast, the logic blocks in [86] include different types of cells, each with set communication patterns. Finally, both designs are restricted to asynchronous applications that do not use arbiters.

The most recent design is the Achronix asynchronous FPGA [10]. It is a high throughput commercial FPGA which claims to be the fastest FPGA on the market.

Achronix can run at 1.5 GHz for the 65nm Speedster FPGA family. The design of Achronix is based on the FPGA described in [86] (spin out of the work by the same group). It maximises throughout but at the expense of power consumption.

2.4.2 Coarse-grained Asynchronous Reconfigurable Computers

This section describes various implementations of asynchronous reconfigurable architectures. The replacement of global clock signals with local handshaking combined with potential power, performance and robustness benefits make asynchronous logic attractive for the design of reconfigurable architectures. Different asynchronous design techniques have been effectively used to create a wide variety of reconfigurable architectures. In [90], an asynchronous reconfigurable architecture for cryptographic applications is presented. It uses homogeneous coarse-grain cells with 8-bit wide data. The functional units are composed of a series of various operation modules such as adder and XOR. This design is custom built for cryptographic applications and uses dual-rail four-phase handshaking protocol and island-style interconnect topology in order to achieve better performance for their applications than on a synchronous FPGA.

A hybrid architecture called Tartan is presented in [91]. It is composed of a hierarchical coarse-grained asynchronous Reconfigurable Fabric (RF) and a Reduced Instruction Set Computing (RISC) CPU core. Tartan uses the spatial computation model where applications developed in C language are compiled and translated by separate tools into RF logic netlist. The structure of the RF is based on the synchronous Piperench architecture [92]: the basic processing elements in the RF are ALU based with 8-bit data width. These processing elements combine to form a stripe: 16 stripes form the RF page and a grouping of 4 x 4 pages constitute a cluster. The intra-cluster communication is performed through switch boxes and the clusters are integrated with a dynamically routed asynchronous Network on Chip (NoC) based on the network design presented in [93]. The synchronous CPU and asynchronous RF communicate through a 96-bit wide bus using mixed-timing

FIFOs. The asynchronous RF is based on bundled data encoding codes. Tartan provides a successful example of applying asynchronous design techniques to a synchronous reconfigurable architecture to reduce energy consumption.

2.5 The RICA Architecture

RICA [4], which DRAP is based on, is a dynamically reconfigurable array, designed at Edinburgh University that fills the gap between FPGAs and ASICs. RICA based processors are coarse grained reconfigurable computing fabrics, consisting of a heterogeneous array of programmable cells on a programmable interconnect network. A heterogeneous array was preferred over a homogeneous one, because, as shown in Section 3.2 of [74], it is more area efficient, without sacrificing flexibility.

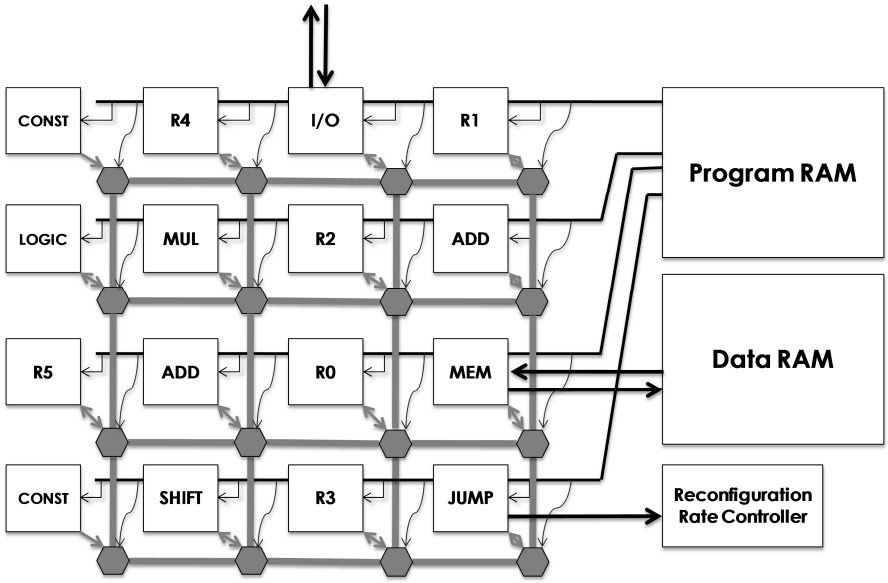


Figure 2.11: Simplified example of a RICA based architecture.

A diagram of a simplified RICA array is shown in Figure 2.11. The operational cells are chosen to match the data width and functionality of RISC instructions in a typical C compiler. They can be combined through the reconfigurable interconnect network to perform more complex instructions in a single configuration context - called a

step. A configuration step persists for the time to allow the sequence of connected cells which form the critical path to complete, and then the next configuration step is loaded. The main features of RICA are as follows:

1. RICA uses the concept of distributed registers—a significant fraction of the instruction cells are registers.
2. The array uses a Harvard memory architecture (to maximise bandwidth)—the program memory and data memory are separate. In many application domains, the array can also have special-purpose stream memories (line buffers) which further increase the on-chip bandwidth.
3. The structure of the core allows complex datapaths to be constructed between the available operational cells.
4. The array is in control of its own reconfiguration: the JUMP cell (Figure 2.11) provides access to the program counter and allows a mapped datapath to influence program control flow [4].
5. To account for the varying critical path of each configuration step, a Reconfiguration Rate Controller (RRC) is used to control the length of time (number of master clock cycles) for which the configuration step persists. This value is stored as part of each configuration context. The RRC connects to all the synchronous cells such as registers in RICA. The state of these cells is updated and the configuration context referenced by the program counter is loaded only when the RRC expires. The program counter may refer to the same step as had just ended, in which case that configuration context persists for another iteration, without incurring any transactions from program memory, or any reconfiguration delay.

2.6 Summary

This chapter provided an overview of the field of asynchronous circuit design, described the main requirements for current and future mobile applications, and explored the different programmable hardware classes. The chapter also presented an

overview of asynchronous reconfigurable architectures and described the RICA architecture in more detail. From this chapter, the following is concluded:

- The field of asynchronous design has been growing and there have been new innovations in tools as well as new reconfigurable asynchronous architectures.
- The trend in designing reconfigurable asynchronous architectures is to start with an existing synchronous design and use it as a stepping stone to a new asynchronous architecture. This is done by applying asynchronous design techniques to the base architecture in order to improve throughput and/or reduce power consumption. In the examples shown in Section 2.4, dual-rail encoding techniques were applied on FPGAs and also on a coarse-grained architecture to increase their throughputs. Bundled-data encoding was used on a coarse-grained architecture to reduce its energy consumption.
- CGRAs are more suited for the growing multimedia and streaming applications than microprocessors and FPGAs. They are also an ideal candidate for applying asynchronous techniques to them due to their large clock trees yet smaller number of functional units compared to FPGAs (Section 2.3).
- Among the large number of existing CGRAs, there are few solutions that support all the requirements of mobile applications for high performance, programmability and flexibility, and energy efficiency [72][74]. The RICA architecture is one such solution. Additionally, the coarse-grained nature of the RICA architecture along with its large clock tree make it a great candidate for applying asynchronous design techniques to it in order to improve energy efficiency and scalability. Because of these properties of RICA, it was chosen as the base architecture for DRAP.

Chapter 3

DRAP Overview

As described in Chapter 1, there is a need in future portable SoC designs for high computational performance along with low power consumption and a high degree of flexibility and programmability. Although ASICs are currently the prime choice for such designs, they are expensive to design and have high levels of inflexibility. This makes them unsuitable for such rapidly changing requirements and markets. At the same time, programmable solutions such as microprocessors offer high programmability but at the expense of a low performance and a high power consumption and area. Reconfigurable datapath solutions such as FPGAs offer flexibility but suffer from high power consumption and are difficult to program. As shown in Sections 2.3 and 2.5, RICA has successfully demonstrated itself as a promising solution for achieving a balance between ASICs on one hand and FPGAs and microprocessors (such as mobile CPUs and VLIWs) on the other in order to bridge the gaps in cost, performance and power consumption between these alternatives.

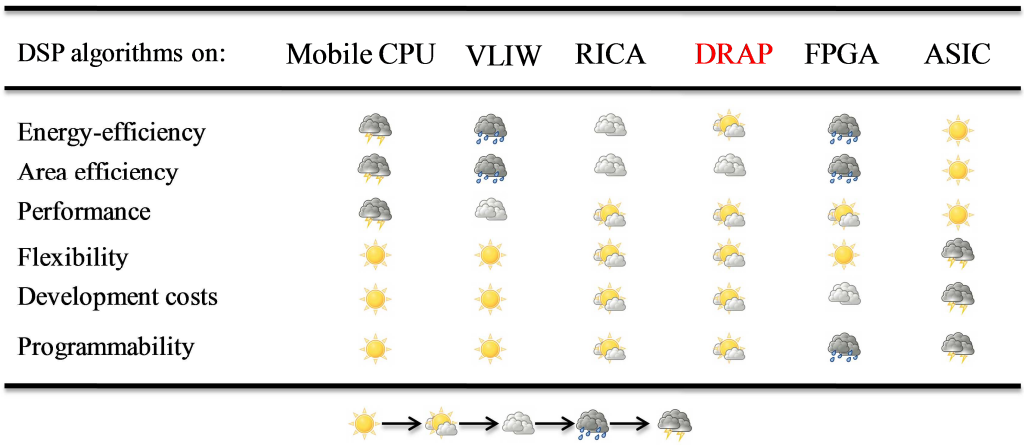


Figure 3.1: Forecast for executing DSP algorithms on various architectures.

Independently, asynchronous logic has made continuous progress in the last decade: several asynchronous processors have been designed [10][86][87][90][91], and CAD tools have been developed [11][17]. Asynchronous logic is a way to design digital systems without clocks. Global synchrony in digital design is replaced with local synchronisation among parts that exchange data. Asynchronous design is also commonly viewed as a low-power design method [5]. The elimination of a global clock combined with other potential power benefits makes asynchronous logic an appealing method for the design of reconfigurable systems. Since two of the main advantages of asynchronous logic are low power consumption and ease of pipelining, it seems natural to apply asynchronous technology to the RICA design. The result is a Dynamically Reconfigurable Asynchronous Processor called DRAP. DRAP is based on the RICA architecture and uses asynchronous design techniques in order to achieve a reduction in power consumption over leading processors while maintaining a high degree of programmability and flexibility (Figure 3.1). This makes DRAP a prime candidate to target future mobile applications.

This chapter introduces the DRAP design and assesses the advantages and disadvantages gained by its structure.

3.1 Architecture overview

DRAP is based on the RICA architecture as described in Section 2.5. As such, the top-level view of the DRAP architecture along with the interconnect mesh, memory architecture, and the nature and types of cells are based on RICA.

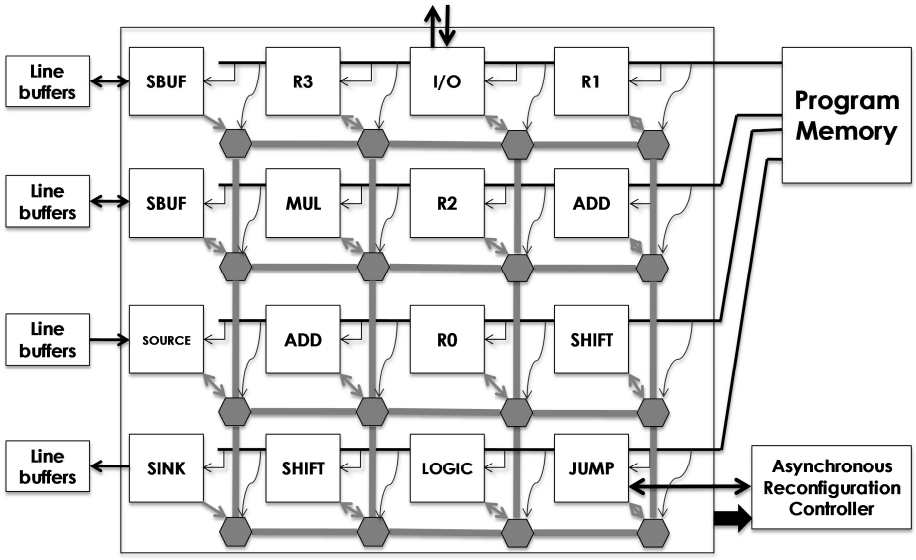


Figure 3.2: Overview of the DRAP architecture. R0 to R3 are asynchronous register cells. The cells in this figure are distributed randomly.

DRAP consists of a heterogeneous array of coarse-grained asynchronous operational cells. Figure 3.2 shows an abstract view of the architecture. The operational cells are interconnected through a network of programmable switches to allow the creation of datapaths. The configuration of the operational cells and interconnects are changeable to execute different blocks of instructions.

Because of the coarse-grained nature of the architecture, the size of the configuration context is small enough to allow reconfiguration times in the order of nanoseconds. As a result, even control tasks can be performed directly on the reconfigurable array, by rapidly switching between different configuration contexts. Additionally, a configuration context does not have to be a free-standing circuit. A kernel of an

algorithm that is too large to fit into a single configuration context can be split into a sequence of steps.

DRAP uses a Harvard memory architecture where the program and data memory are separate. The array contains Interface cells (source, sink, sbuf) – cells which act as an interface to special-purpose stream memories (line buffers). These are used in many application domains to increase the on-chip bandwidth. The program memory contains the configuration bits that control both the operational cells and the interconnect switches. Data is passed in and out of the array via the Interface cells.

A basic set of cell types was provided, which closely match typical RISC instructions (e.g. add, multiply, shift, logic, etc.), along with some special-purpose cells such as a program flow control (jump). These cells are often referred to in the text by their corresponding instruction name, expressed in block capitals (e.g. ADD). In some variants of the core, certain primitives were combined (e.g. add and comp → addcomp, instruction mnemonic ADDCOMP). The array size and individual cell counts are design variables.

The operational cells are designed using 4-phase handshaking protocol and bundled data encoding. Section 2.1.5 provides a comparison between bundled data encoding and dual-rail encoding methods. Dual-rail is truly delay-insensitive but results in large cells since each bit is represented by two wires. Additionally, dual-rail operational cells would require a larger interconnect structure than an equivalent cell using bundled data encoding to route the extra wires. Bundled data uses fewer wires and results in smaller cells and interconnects and hence was preferred in the DRAP design.

Section 2.1.3 describes the several signalling protocols used to encode handshaking onto specific control wires. The two most common protocols are two-phase and four-phase signalling. Four-phase uses simpler and smaller hardware than two-phase as it should be sensitive to only one edge. On the other hand, two-phase signalling has the potential to be faster and more power efficient than four-phase because it uses each

transition in a handshake. However, this is not the case with current tools such as Balsa[50] and TiDE [48] which favour four-phase signalling. As a result, four-phase signalling was the choice in the operational cell design.

The Asynchronous Reconfiguration Controller (ARC) extracts information from the handshaking between certain operational cells of a mapped datapath to indicate when a configuration context has finished. Similarly to RICA (see Section 2.5), DRAP uses a special cell, the JUMP cell, to control its reconfiguration. JUMP provides access to the program counter and allows the datapaths to influence program control flow. Finally, DRAP has asynchronous register cells distributed around the array. They are to be used as delay elements, for pipelining or to save data between configuration contexts. The asynchronous operational cells, interconnect structure, and support hardware were all implemented as Verilog modules.

Figure 3.3 shows how DRAP is programmed. The software flow of DRAP, explained in more detail in Section 7.2, is based on that of RICA. Applications are written in a high-level software programming language (C), and tools automatically convert this into configuration contexts (routed netlists). For the first step, the tools take the high-level code, along with a description of the array characterised as a Machine Description File (MDF), and transform it into an intermediate assembly language. The tools automate the process of extracting Instruction-Level Parallelism (ILP) from the basic blocks of the program, and then mapping the instructions of each basic block to operational cells in the array (abstract netlist). Large blocks are split into sequences of smaller blocks if there are insufficient operational cells. A routing tool (mapper) then configures the reconfigurable interconnect, and adds the FINISH signal.

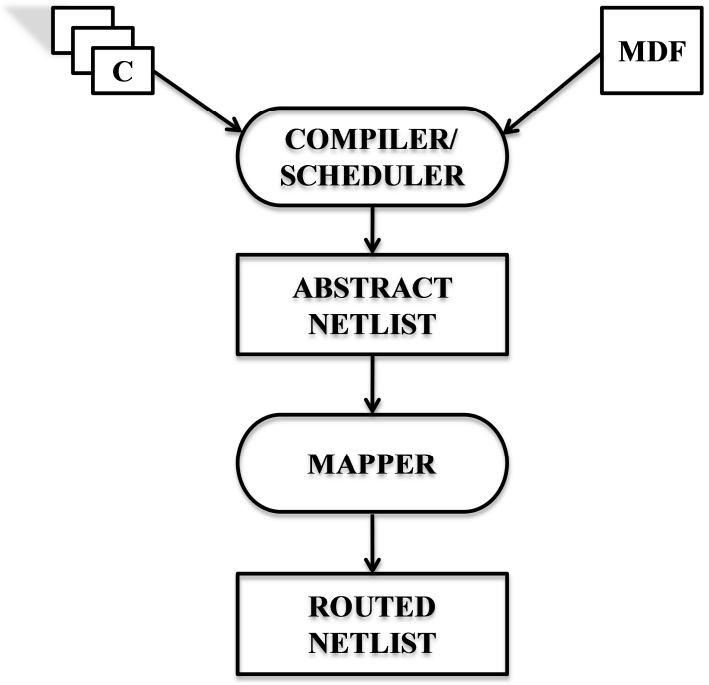


Figure 3.3: Software flow for programming DRAP starting from high-level C program.

3.2 Characteristics of DRAP

3.2.1 Energy driven consumption

Asynchronous design techniques are based on local communication among units. Communication and synchronisation among the units is implemented by handshaking. There is no concept of global time and hence no global clock is used. Asynchronous logic inherently implements the synchronous equivalent of perfect clock gating. Parts of the array that do not contribute to the computation are automatically turned off and have no switching activity. Furthermore, within a mapped datapath, the nature of the asynchronous cells allows the input to pass through the combinatorial part of a cell only after it has become valid. Hence invalid data does not cause unnecessary activity and waste energy (Figure 3.4).

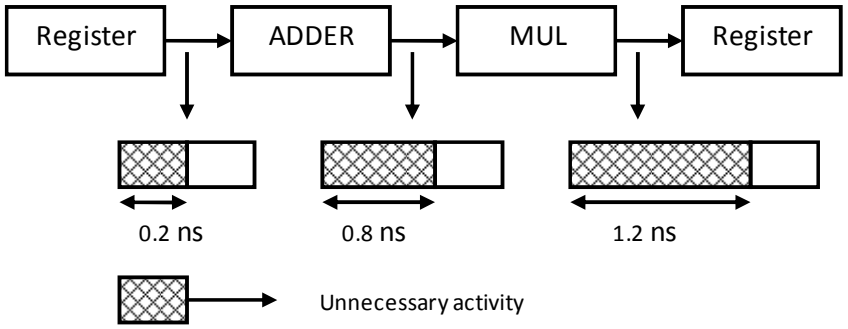


Figure 3.4: Invalid data causing unnecessary activity in the combinatorial cells of a synchronous design, leading to wasted energy.

The drawback of using handshaking to communicate betweens cells (as opposed to a global clock) is the area increase associated with local control structures: latches within each cell at its inputs and/or output and additional logic in the interconnects to deal with the request and acknowledge signals. In 32nm libraries and below, leakage power becomes increasingly dominant over switching power. Therefore, the leakage power associated with an area increase could offset the savings in switching power - leading to an overall increase in power.

Table 3.1: Area comparison of cells with asynchronous control (0.13µm process technology).

Cell	Logic area (μm^2)	Control area (μm^2)	Latches area (μm^2)
MUL	6,618.24	221.18	770.69
ADDCOMP	4,456.51	231.55	749.95

However because of the coarse-grain nature of the DRAP array, the increase in area is relatively small. The increase is also offset by the elimination of the clock tree and also by a reduction in number of registers needed for pipelining since the asynchronous cells already provide a certain level of pipelining. If the area of latches within the asynchronous cell is ignored, the asynchronous control area of an 18-bit MUL and ADDCOMP cells forms 3.3% and 4.9% of the total cell area (latches not included) respectively (Table 3.1).

Additionally, a significant area reduction comes from an increase in routability: DRAP offers implicit pipelining (see Section 7.1.2) and hence requires a smaller number of interconnect channels to route pipelined applications than RICA. Explicit pipelining (see Section 7.1.1) on RICA requires connecting additional registers into the datapaths, which increases demand and pressure on the reconfigurable interconnect. On DRAP, the registers used for implicit pipelining are already a hard-wired part of the datapaths. For example, for an application like the bilinear demosaic filter [94] (described in more detail in Section 8.1), RICA needed a 5 channel interconnect to comfortably route the pipelined application on the array. Interconnect channels are described in more detail in Section 5.2.1. For DRAP, a 4-channel interconnect was enough to route the same algorithm.

As can be seen from Table 3.2, for a 5-channel switchbox with seven out of its 20 channels pipelined, the area of the DRAP switchbox is 25% larger in area than that of the RICA one. However, that area difference is reduced to 4.5% when a 4-channel switchbox for DRAP is used.

Table 3.2: Comparing area of interconnects and number of configuration bits for asynchronous (DRAP) and synchronous (RICA) designs.

Design	Description	Area (μm^2)/switchbox	Number of Configuration bits
RICA	5-channel, 15x15 array – 7 registers (7/20 pipelined switchbox channels)	21,000	18,660
DRAP	5-channel, 15x15 array – 2 registers (7/20 pipelined switchbox channels)	28,000	16,520
DRAP	4-channel, 15x15 array 2 registers (7/16 pipelined switchbox channels)	22,000	13,820

3.2.2 Implicit Pipelining

As mentioned above, each asynchronous cell contains latches at its inputs and/or output. This provides full pipelining within datapaths. In RICA, a large number of register cells is needed for pipelining. In DRAP, the latches that exist inherently within the asynchronous cells and interconnect are used for pipelining. The asynchronous register cells are primarily used for storage and can also provide pipelining. As a result, DRAP requires significantly less pipelining dedicated registers compared to RICA. This leads to a saving in area and configuration bits. For the latest design, a 15x15 DRAP array contained 450 asynchronous register cells (two per switchbox). An equivalent RICA design required 1575 register cells (seven per switchbox) to run the same applications.

A characteristic of implicit pipelining in bundled data asynchronous design is that it cannot be turned off. As a result, steps that don't need pipelining are still pipelined. These steps will run at slower speeds than they would with no pipelining because of delays introduced by handshaking. However, for DRAP's targeted applications (mobile applications), most of the execution time (over 95%) is spent in kernels. This other work [95] shows one method to control implicit pipelining (turn it off or on). However, for DRAP's purpose, using such a method is unnecessary because if a step is not a kernel (i.e. does not require to be pipelined), the configuration loading dominates the execution time of the step.

3.2.3 Scalability

Most of the mobile applications use large kernels. This makes it desirable to use a larger RICA or DRAP core i.e. to increase the number of operational cells in order to run such applications at higher throughputs by mapping the entire kernel onto one configuration context. When a RICA core gets bigger, the number of sequential elements (primarily registers) in the array must be increased significantly in order to maintain routability and pipelining. The clock tree therefore takes up a larger

percentage of the area and power consumption of the core. By using asynchronous design techniques, DRAP eliminates the clock tree and replaces it with local handshaking which implements synchronisation among the operational cells of an asynchronous datapath irrespective of its length. This makes DRAP inherently more scalable.

However, no global clock means there is no notion of how much time a configuration context must persist for. A scheme was devised to indicate when a configuration context has terminated and hence when the next one can be loaded. The ARC module was introduced to indicate when a step has concluded its work. It extracts a FINISH signal from the handshaking signals of certain cells and feeds it to the JUMP cell. The scheme was designed in a way that maintains the inherent scalability of DRAP. The design of the ARC module and the JUMP cell is discussed in more detail in Chapter 6.

Furthermore, datapaths on DRAP benefit from implicit pipelining via the handshaking latches - configuration contexts do not need to be explicitly pipelined; therefore much fewer registers are needed. This makes this scheme more scalable than having a global clock - both in terms of power and area overhead.

3.2.4 Reduced program size

As mentioned in Sections 3.2.1 and 3.2.2, certain aspects of the DRAP design allowed a reduction in the program size compared to RICA. Implicit pipelining in asynchronous logic means that DRAP provides full pipelining for most mapped datapaths automatically. On the other hand, RICA requires a large number of register cells to be able to pipeline a kernel step. That makes datapaths in RICA longer and harder to route on the array. As a result, RICA also requires more channels per interconnect structure to pipeline and route applications with large kernels than DRAP does. For a 15x15 array, DRAP used 71% fewer registers than RICA and a 4-channel interconnect compared to RICA’s 5-channel to pipeline and route the kernel

of the Bilinear application. This translated to a reduction of 11% and 26% in configuration bits for a 5-channel and 4-channel DRAP switchbox respectively.

3.3 Summary

In this chapter, the DRAP design was introduced. It is made up of a heterogeneous array of coarse-grained asynchronous operational cells. The cells were designed using 4-phase bundled data asynchronous logic because it allows for simpler and smaller hardware. DRAP contains a basic set of cell types, such as add, multiply, and shift, which closely match typical RISC instructions along with some special-purpose cells such as a program flow control cell called CJUMP.

An overview of how DRAP is programmed was presented. Applications are written in a high-level software programming language such as C, and the DRAP tools automatically convert this into configuration contexts.

Finally, the characteristics of the DRAP architecture were listed and assessed with some results provided. These are summarised as follows:

- Energy-driven consumption: Asynchronous logic inherently implements the synchronous equivalent of clock gating, where parts of the array that do not contribute to the computation are automatically turned off and have no switching activity. This allows DRAP to reduce its power consumption compared to its synchronous counterparts. There is an area penalty for using asynchronous logic but techniques to counter it are presented.
- Implicit pipelining: The DRAP cells inherently contain handshake controlled latches at their inputs and/or outputs. This resulted in an ease of fully pipelining applications and an increase in routability on DRAP.
- Scalability: The DRAP array contains no global clock signal and hence is more scalable in terms of area and power than its synchronous counterparts. The mechanism that controls reconfiguration in DRAP was designed to be scalable.

- Reduced program size: The implicit pipelining and increased routability provided by DRAP means fewer interconnect channels and pipeline dedicated registers are needed. This allows DRAP savings of up to 26% on configuration bits when compared to an equivalent RICA design.

Chapter 4

Asynchronous Operational Cells

In Chapter 3, an overview of the DRAP system architecture was presented and the choices made in designing its asynchronous operational cells were discussed. The following asynchronous techniques were chosen for the DRAP design: 4-phase handshaking protocol and bundled data encoding. This chapter describes the asynchronous operational cells used in DRAP, their properties, and how they were designed. It also presents and compares different methods of designing the asynchronous cells.

4.1 Overview of Cell Design for DRAP

The asynchronous cell array in DRAP is heterogeneous and coarse grain, and each operational cell is limited to a small number of operations as listed in Table 4.1. Because the operational cells are specific in function and hence small in area, the overhead of increasing the size of a DRAP array is dominated by the extra interconnect area required for the additional cells. The use of heterogeneous cells allows the array to be tailored to a specific application domain by adding extra

operational cell types for frequent operations in that domain. Additionally, it was shown in [74] that a heterogeneous cell array is more area efficient than a homogeneous one, without sacrificing flexibility.

As with the RICA architecture, all the DRAP cells have at most one output and most have two inputs only. The operational cell interconnect was designed for a 2-input and 1-output cell. This makes it easier to create a more efficient interconnect structure and reduces the number of configuration bits needed. Operational cells which require more than two inputs were spread over several interconnects.

As for the granularity of the cells, two basic DRAP arrays were designed: one consisting of 18-bit operational cells and the other of 32-bit ones. Each cell is connected to a switchbox. The switchboxes are connected in a simple 2-D grid, with 18/32-bit unidirectional interconnect. Each of the four directions has input and output channels, connecting the cell to its neighbours. The design of the interconnect structure is described in more detail in Chapter 5.

As shown in Table 4.1, a basic set of asynchronous cells was provided. These resemble RISC instructions like addition, multiplication, shift, and logic. This denomination is not fixed and the scope of the operations of the asynchronous cells can be expanded in the future.

Memory elements and asynchronous registers are defined as standard instruction-cells and are distributed throughout the array. This allows a higher degree of instruction-level parallelism (bandwidth) than architectures with a fixed register file.

Special instruction cells include the JUMP cell which acts as an instruction-controller responsible for managing the program counter and the interface to the program-memory, as well as controlling when certain cells in the array can start computing. The interface with the data-memory is provided by the SBUF cells (high-bandwidth stream buffers). Communication with external logic is done through the SINK and SOURCE cells (FIFOs to external stream data sources/sinks). All these special

instruction cells provide an interface to synchronous hardware, but they themselves are asynchronous.

Table 4.1: Possible operational cells and their supported operations.

Asynchronous Operational cell	Supported Operations
ADD	Addition, Subtraction
MUL	Multiplication (Signed, Unsigned)
DIV	Division (Signed, Unsigned)
REG	Registers
SHIFT	Shifting operation
LOGIC	Logic operation (XOR, AND, OR, etc)
COMP	Data comparison
MUX	Allows simple branches to be taken
JUMP	Branches (and sequencer functionality)
SOURCE, SBUF, SINK	Stream memory /line buffers

4.2 Synthesis of Asynchronous Operational cells

Synthesis of the cells was done using the automated decomposition tool TiDE from Handshake Solutions [48]. A high-level concurrent programming language called Haste described the operational cells and then synthesised in two stages to a Verilog netlist based on cells from a standard-cell library. The first stage translates the Haste code into an intermediate Handshake Circuit in a transparent, syntax-directed process and the next stage maps the Handshake Circuit to a structural Verilog netlist and for initial circuit-level optimisation [48]. The resulting cell includes two parts (Figure 4.1): the datapath which contains flip-flops, latches and combinatorial logic blocks, and the control which contains matched delay chains and asynchronous logic with feedback loops. The tool also allows the import of non-Haste combinatorial functions. It was therefore possible to develop and optimise some logic functions in

Verilog and import them into the Haste designs. More information on the TiDE tool can be found in Appendix C.

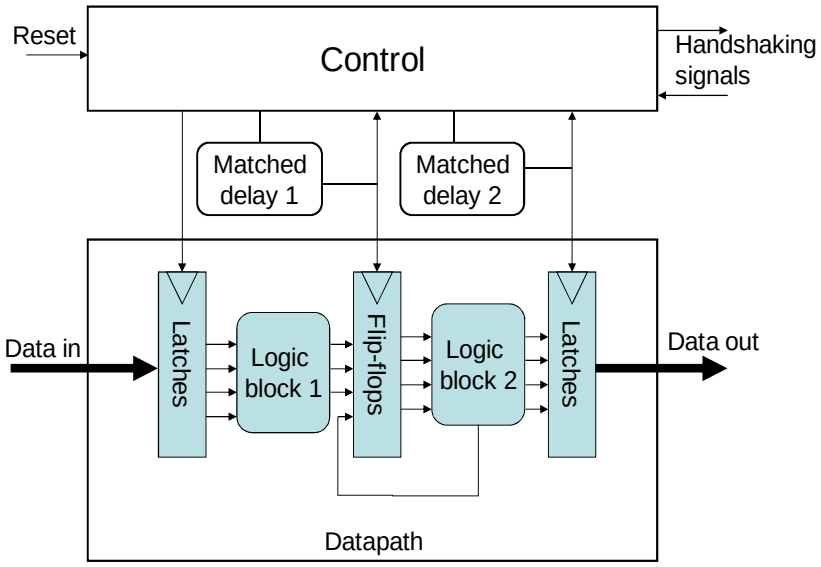


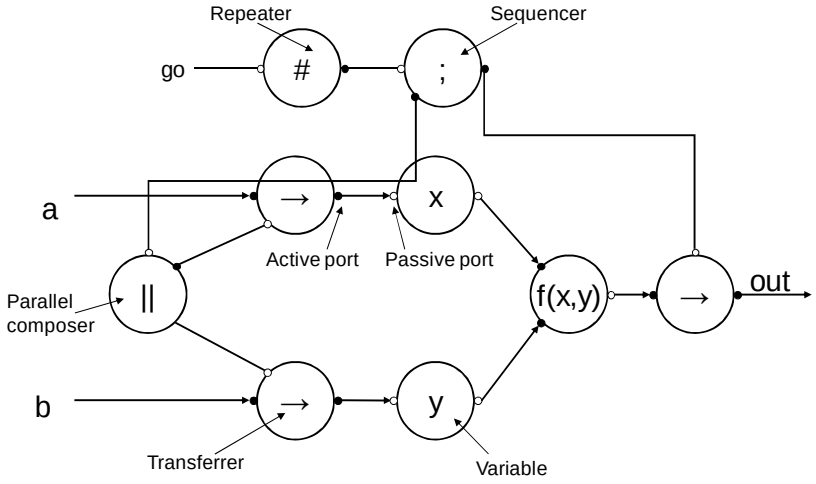
Figure 4.1: Asynchronous circuit block diagram showing the Control and Datapath parts.

Figure 4.2 shows a Haste description and the equivalent handshake circuit graph of a general 2-input, 1-output operational cell. Handshake components communicate through ports. A channel connects one active port, which initiates communication by sending a request to one passive port, which responds with an acknowledge. The passive port *go* is the activation port of the handshake circuit. When the cell is activated, a request signal is sent down both input channels *a* and *b*, and data from the channels will only be transferred to the variables once both channels have acknowledged their respective requests.

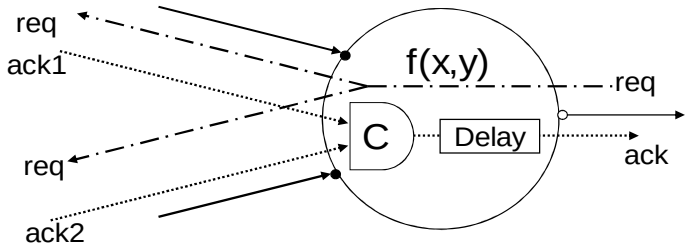
In conventional synchronous design, invalid data propagate through the inputs of cells and cause unnecessary activity. Asynchronous logic implicitly implements fine-grain “clock gating” by automatically turning off unused parts of the design. Additionally, each asynchronous cell will wait for the data at its input to become valid before letting it through. Hence unnecessary computation in active cells is eliminated and unused cells have no switching activity.

```
1. Operational_Cell: main proc(a,b? chan int32 & out! chan int32)
2. begin
3. x, y: var int32
4. forever do
5. | // bar that separates auxiliaries from body of code
6. a?x || b?y // write both input channels into variable in parallel
7. ;out! f(x,y) // generate the function output and read onto channel
8. od
9. end
```

(a)



(b)



(c)

Figure 4.2: (a) Haste description of an operational cell (b) Equivalent handshake circuit (c) A closer look at the control signals inside a handshake component: the C-element synchronises the incoming acknowledge signals and the resulting signal passes through a delay matching the critical path of the function.

4.3 Cell Design Variations

Figure 4.2a shows a Haste description of a functional cell. Line 6 of the code, *a?x || b?y*, activates writing to both the input channels in parallel. The Haste language is rich, and there are several ways of performing the same asynchronous control task. In this subsection, some of the choices available to the Haste designer are discussed.

There are different ways of reading the inputs to an asynchronous channel. As an example, there is another way to write the values of the latches *a* and *b* to the input channels. To enable a faster implementation, the parallel input actions are rewritten to one big input action of the following form: *<<a,b>> ? <<x,y>>*. This new form has only a single acknowledge signal, covering both inputs. The downside of this implementation is that it can slow down or even cause deadlock in cases where the datapaths leading up to both inputs are interconnected.

The tool provides other options. The designer has a choice to map the variables in the Haste code into either latches or flip-flops. Using flip-flops over latches should reduce the possibility of glitches in the design. However, with flip-flops, the design would be larger and potentially slower.

The code in Figure 4.2a shows the function being called and read directly onto the output channel (line 7 of code). This means the functional cell being described has latches/flip-flops for its inputs only. Another way of designing the cell would be to assign the result of the function to an output variable labelled *z* (*z = f(x,y)*), and *z* being read onto the output channel (*out!z*). In this case, the functional cell would have latches/flip-flops for its inputs and its output. This new form requires one more latch/flip-flop. However, it could be faster since the inputs don't have to wait for the output handshake to terminate before responding to new requests.

Because of the transparent, syntax-directed nature of the tool, a small change in the Haste program could directly transfer to a significant loss or gain in area, speed, or power consumption. As a result, all combinations of the choices mentioned above

were tested on a 32-bit ADDCOMP cell. The functional part of the cell was similar. The variations of the cell are summarised below. Each version of the cell was mapped twice, once using latches and once using flip-flops:

1. V1: write to input using *<<a,b>> ? <<x,y>>*. Result of computation assigned to output latch/flip-flop.
2. V2: write to input using *a?x || b?y*. Result of computation assigned to output latch/flip-flop.
3. V3: write to input using *<<a,b>> ? <<x,y>>*. Result of computation read out directly without the need of an output latch.
4. V4: write to input using *a?x || b?y*. Result of computation read out directly without the need of an output latch.

Table 4.2: Comparing area, delay, power, and energy of different versions of an asynchronous operational cell (ADDCOMP).

Design	Area (μm2)	Delay (ns)	Time (ns) (2000 iterations)	Power (mW)	Energy (nJ)
V1 (latches)	7,314	5.4	11,099	0.93	10.3
V1 (flip-flops)	9,669	8.5	17,205	0.62	10.7
V2 (latches)	7,428	6.1	12,420	1.29	15.9
V2 (flip-flops)	9,706	6.5	13,236	1.11	14.7
V3 (latches)	9,367	5.8	11,755	0.70	8.3
V3 (flip-flops)	11,335	8.2	16,622	0.48	7.9
V4 (latches)	9,453	6.5	13,203	0.67	8.9
V4 (flip-flops)	10,971	6.4	13,116	0.66	8.7

A datapath made of four ADDCOMPs connected in series was designed for each cell variation. An identical set of 2000 inputs were tested on the datapaths and the area,

delay and power values measured. The area is measured per cell and was obtained by the TiDE tool (pre-routing). The datapaths are implemented using a UMC 0.13-µm technology. The power and speed were found using post-layout simulations on PrimePower from Synopsys. All these power estimations were measured at 1.2-V operating voltage.

The results are summarised in Table 4.2. As expected, the designs mapped using flip-flops as opposed to latches are larger in area (between 14% - 24% larger) and almost always slower. However, because they reduce switching activity, the energy consumed is almost always lower.

Comparing the way input channel actions are implemented, the first technique (*a?x || b?y*) always results in slower cells than the second technique (*<<a,b>> ? <<x,y>>*). However, as mentioned above, there are potential cases where the second technique could lead to deadlock and so it was not used in the design of the DRAP array. This rules out designs V1 and V3.

Design V2 compared to design V3 is slightly faster but consumes more power and energy and was hence not used in the design of the array. The design that was used in the DRAP array is based on V4 with latches rather than flip-flops due to the reduced area.

4.4 Description of the Operational Cells

A brief description of the cells in Table 4.1 is presented below. Appendix A contains more details of the operational cells in the sample DRAP. All the cells were designed to be asynchronous and each input and output signal has corresponding handshake signals (request and acknowledge) associated with it.

ADD (2-inputs; 1-output):

This cell supports addition and subtraction operations. The cell can be extended to support complex addition/subtraction where the input data is split between the real

and imaginary parts (e.g. a 32-bit DRAP would have a 16-bit imaginary part and a 16-bit real part).

MUL (2-inputs; 1-output):

This cell supports signed and unsigned multiplication. Similar to the ADD cell, it can be extended to support complex multiplication.

COMP (2-inputs; 1-output):

This cell compares its two inputs and outputs the result of the comparison generated as a data signal.

ADDCOMP (2-inputs; 1-output):

This cell performs the functions of both the ADD and the COMP cell.

DIV (2-inputs; 1-output):

This cell support signed or unsigned integer division.

LOGIC (2-inputs; 1-output):

This cell performs standard bit-operations such as AND, OR, NAND, NOR, XOR, NOT, as well as bit-reversion and 2's complement negation.

SHIFT (2-inputs; 1-output):

This cell performs logical and arithmetic left/right shifting.

MUX (3-inputs; 1-output):

This cell receives 3 inputs: Two data signals in1 and in2, and a select signal (which is often the result of the comparison coming from a COMP or LOGIC cell). Depending on the select signal, it routes either in1 or in2 to its output. In addition to being a

multiplexer in hardware, this cell acts as a conditional-move operation from a software point of view.

JUMP (2-inputs; 1-output):

The JUMP cell acts as the instruction-controller and manages the Program Counter. The program counter is given to the Program Memory controller to retrieve the configuration of the cell for the current steps. Additionally this cell controls the REG and interface cells (SOURCE, SINK, SBUF) by indicating to them when it is safe to start computing. The JUMP cell is discussed in more detail in Chapter 6 (Section 6.4).

SOURCE (0-inputs; 1-output):

The SOURCE cell provides an interface to the external environment, through FIFOs to a stream data source.

SINK (1-input; 0-output):

The SINK cell provides an interface to the external environment, through FIFOs to a stream data sink.

SBUF (2-inputs; 1-output):

The SBUF cell provides an interface to small, high-bandwidth local data memories, for use in streaming. Each cell acts as an interface to multiple banks of SRAMs.

REG (1-input; 1-output):

The REG cells replace the register file found in a processor, with the difference that the registers are distributed and accessed independently; hence they consume less energy since there is no need to use a large multiplexer to address them. All the REG cells receive a signal from the JUMP cell that indicates it is safe to start computing. The REG cell has four main functioning modes depending on its location in a

datapath. Some of the REG cells had an optional fifth mode where they act as a constant generator. These are as follows:

- REG cell at the start of a datapath: activate output channel.
- REG cell at the end of a datapath: activate input channel and store new value.
- REG cell as a delay element: activate output channel then activate input channel and store new value.
- REG cell for pipelining: activate input channel then activate output one.
- Constant mode (optional): load value from memory and activate output channel.

4.5 Summary

This chapter described the first of two parts of the design of the DRAP hardware: the asynchronous operational cells, their properties, and how they were built were presented. The chapter also presented and compared the different methods of designing circuits, which the asynchronous design tool TiDE allows.

Chapter 5

Asynchronous Interconnect Design

Recently designed asynchronous reconfigurable architectures were presented in Section 2.4. This chapter discusses one of the main challenges in building such architectures: the design of the reconfigurable interconnect scheme. The chapter begins by describing the general interconnect structure for DRAP (Sections 1.1 and 5.2). It then explains the main challenges of designing interconnects for asynchronous reconfigurable circuits. This is followed by an analysis of the design techniques used by other asynchronous reconfigurable architectures to address the challenges. Finally, a novel method for designing interconnects for asynchronous reconfigurable architectures is presented and compared to ones in the literature.

5.1 DRAP Interconnect

The general interconnect structure of an asynchronous reconfigurable architecture can be conceptually modeled after that of an equivalent synchronous architecture. In [96][97][81], different solutions for the circuit design and for the topology of the

interconnect switches for RICA are discussed, along with a comparison between the multiplexer-based crossbar and the island-style mesh found in typical FPGAs [88]. DRAP follows a similar interconnect design as that chosen for RICA, hence why only the results of those discussions are summarised below.

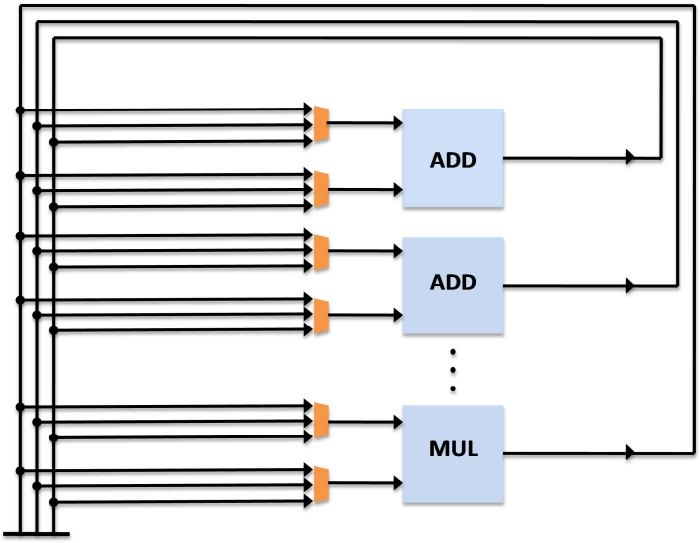


Figure 5.1: A crossbar interconnect scheme using multiplexers [81].

The role of interconnects is to allow the transfer of data from the output of an operational cell to the inputs of other cells in order to form large operational circuits. Ideally, the switching network in a reconfigurable architecture would allow the routing of signals between any two cells in the array at any time. Such an ideal switching network can be implemented using a large multiplexer on each cell input. This multiplexer allows choosing which data to route, and would be connected to the output of all the other cells. This is known as the multiplexer-based crossbar interconnect scheme (Figure 5.1). Although the crossbar scheme is easy to program, it occupies a large area. Furthermore, it limits scalability of an array because the area of the multiplexers increases quadratically as the number of cells in the array increases. Hence, there is a need for an interconnect structure which is small in area but scalable and allows the routing of a wide range of circuits.

The island-style interconnect scheme used in typical FPGAs (Figure 5.2) fits all these requirements. It provides a method of connecting together the cells of an array in a much more area efficient way than the crossbar scheme. Additionally, each cell and its corresponding switchbox are independent: a cell might be inactive in the specific step but its associated switchbox might be used to route a signal belonging to a different cell. On the downside, the island-style scheme requires a larger number of configuration bits and results in larger interconnect delays. In the island-style structure, the interconnect delay is dependent on how many switchboxes a routed signal passes through.

Table 5.1: Comparison between cross-bar and island-style interconnects. [72]

Interconnects for a 64 cell array	Area (μm^2) on 0.13µm	Number of configuration bits	Dela of one connection (output-input, ignoring wire capacitance)
Crossbar	1,640,495	498	0.7 s
Island-style	576,062	678	Variable, average of 5 switchboxes is 2.0 ns

Table 5.1 has been reproduced from [72] courtesy of the original author. It shows a comparison between multiplexer-based crossbar and island-style interconnects for RICA. The comparison is based on a sample array of 64 32-bit cells. As can be seen from the table, the overall area of the island-style interconnect is 64% smaller than the crossbar one. The number of configuration bits required is however increased by 36% and the delay becomes dependent on the routing of the signal and the number of switchboxes it passes through. The delay value given does not include wire delays, which in this case should be much less than the crossbar version, as the metal wires are greatly reduced due to the increased locality. The aforementioned results and conclusions also apply if the interconnects were for an asynchronous system.

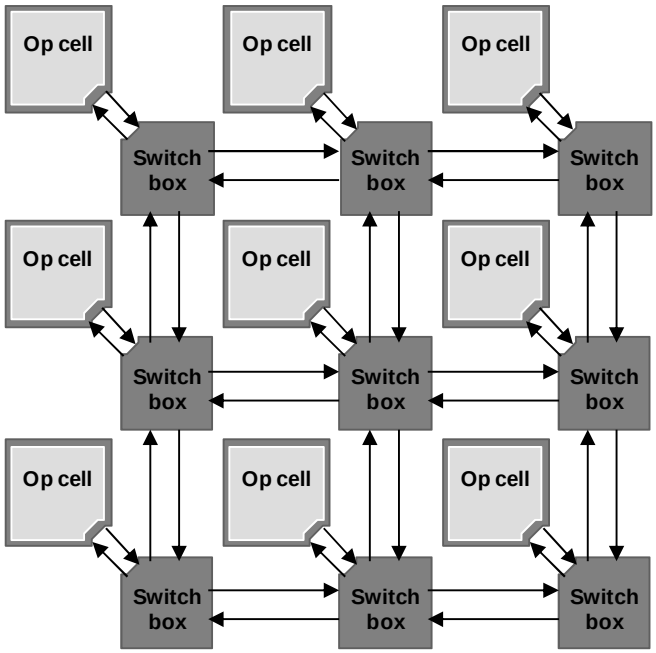


Figure 5.2: Array of operational cells in an island-style mesh-based topology.

The DRAP interconnect design is based on the island-style structure and takes into account the presence of handshaking signals in the operational cells. It also assumes the cells have one output and two inputs and that the output of a cell cannot be looped back to one of its inputs.

Figure 5.3 shows an implementation of the switchbox for DRAP. Each cell is surrounded by four routing switches, one for each side. The switches can be combinatorial multiplexers or asynchronous multiplexers. This will be discussed in more detail in Section 5.2. The signal tracks used are unidirectional, and on each side there is one input and one output asynchronous channel. The routing switch controls the output and its corresponding request signals, and according to its configuration it can route signals that are coming in from other directions to its output. Each switch also receives the output of the current cell to allow routing it to other cells. Furthermore, each operational cell input has a switch that selects which of the four sides should be routed from outside of the box.

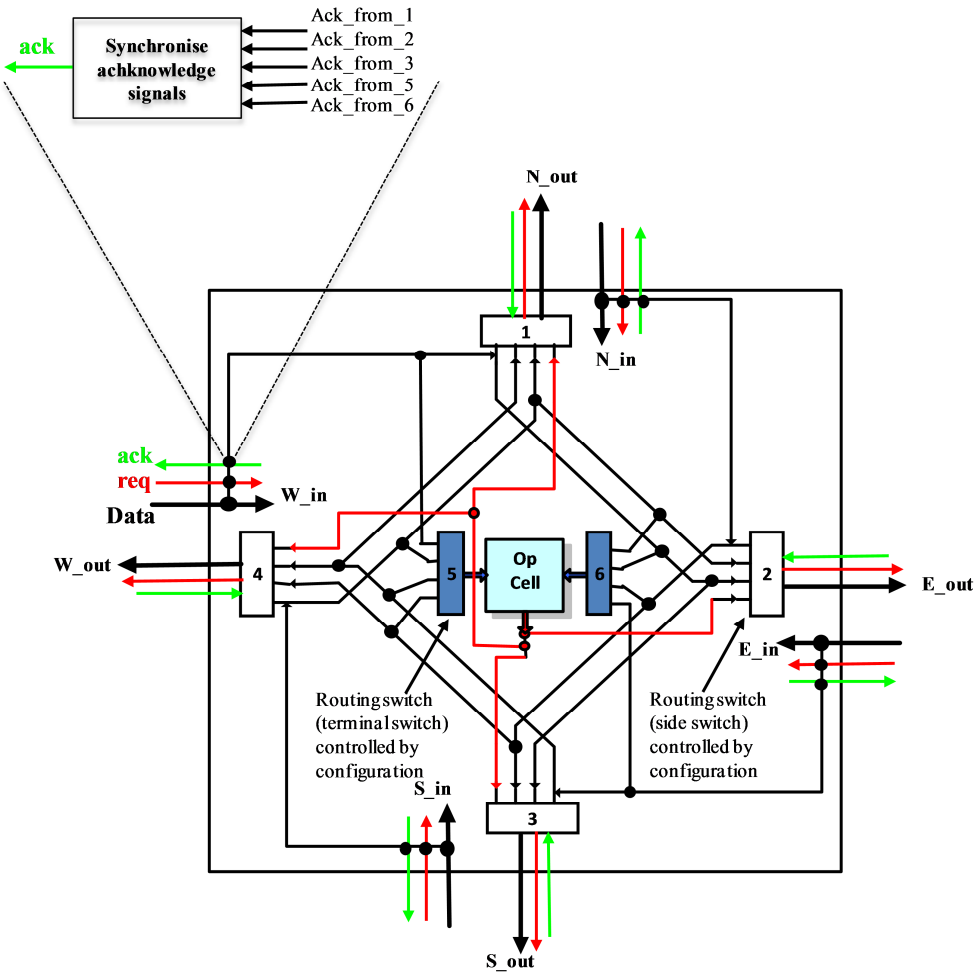


Figure 5.3: The switchbox interconnect with the operational cell inside.

Each switchbox input (N, E, S, W) and its corresponding request signal are connected to several routing switches. As such, it receives a returning acknowledge signal, one from each switch it connects to. These signals have to be synchronised before they are returned to the source of the input signal. Synchronising the acknowledge signals for each switchbox input is a big challenge in the design of interconnects for asynchronous reconfigurable architectures. Sections 5.3 through 5.5 discuss this problem and potential solutions in more detail.

5.2 The Routing Switch

In the previous section, an overview of the interconnect structure for DRAP was presented. The routing switches 1, 2, 3, and 4 (Figure 5.3) are referred to as the side-switches. They surround each asynchronous operational cell and allow it to connect to its neighbours. The routing switches 5 and 6 (Figure 5.3) are located at the cell inputs and are referred to as the terminal-switches. They select which input of the four sides should be routed into the cell.

A combinatorial (non-pipelined) routing switch for both the side and terminal switches (Figure 5.4) was initially designed. The data and request of the input channels are routed to the output channel through multiplexers that use the same select signal while the acknowledge signal of the output of the switch are fanned-out to all inputs.

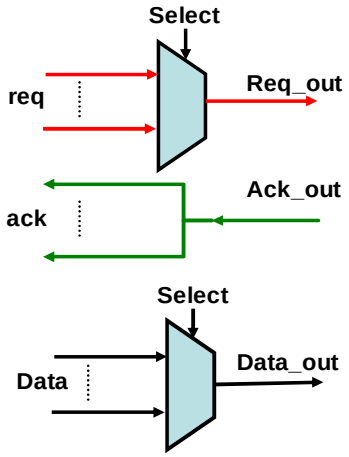


Figure 5.4: Combinatorial design of a DRAP routing switch.

Since a connection between two cells can pass through any number of switchboxes, the interconnect delay is variable. With combinatorial routing switches, the delay to a sender's data and request signals is the sum of the delay of all the routing switches (plus wire delay) connecting the sending and receiving cells. This delay is referred to as the data-request interconnect delay. The delay to the returning receiver

acknowledge signal is the sum of the delays of the blocks which synchronise the acknowledge signals of each input. This delay is referred to as the acknowledge interconnect delay.

The interconnect delay affects both the rising and falling phases of the request and acknowledge handshaking signals. Since DRAP uses four-phase handshake signalling, the data-request and the acknowledge interconnect delays each contribute twice to the overall interconnect delay.

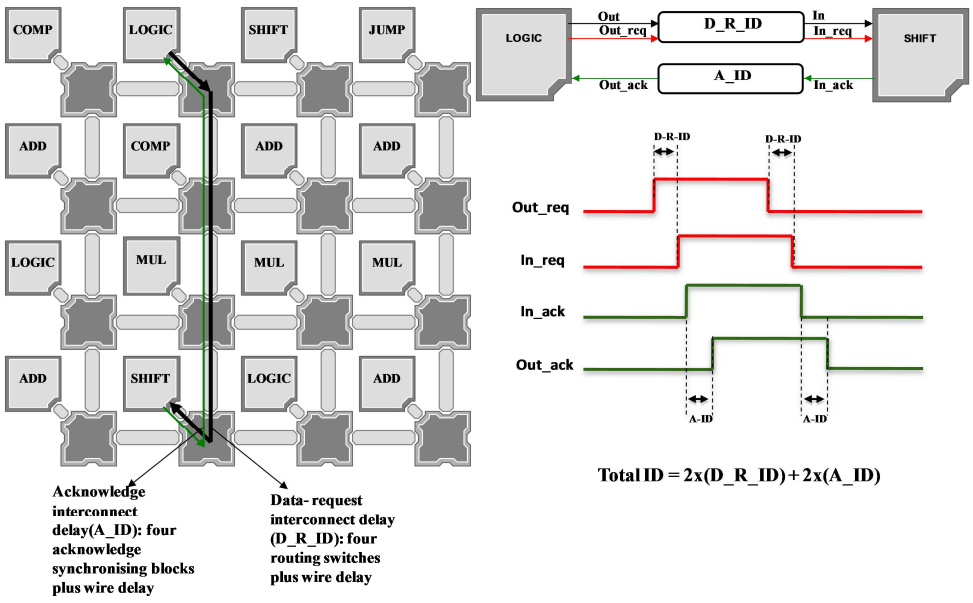


Figure 5.5: Interconnect delays in array using combinatorial interconnects.

Figure 5.5 shows an example of a datapath where the output of a LOGIC cell is connected to the input of a SHIFT cell through four switchboxes. The data-request interconnect delay is that of the wire plus four routing multiplexers and is referred to as D_R_ID. The acknowledge interconnect delay is that of four acknowledge synchronising blocks and is referred to as A_ID. When the output request of the LOGIC cell (Out_req) goes high, the input request port of the SHIFT cell sees the rising transition after a delay of D_R_ID. The SHIFT cell raises its input acknowledge signal (In_ack) in response. The output acknowledge signal of the

LOGIC cell (Out_ack) sees this transition after a delay of A_ID and lowers Out_req as a result. In_req goes low again after a delay of D_R_ID and lowers In_ack. It takes a final delay of A_ID for Out_ack to see the final phase in the handshake. The total delay to the communication between the two cells is:

Equation 5.1: Total interconnect delay as a function of data-request interconnect delay (D_R_ID) and acknowledge interconnect delay (A_ID). This applies to both non-pipelined and pipelined routing switches, but the component values differ (multiple hops vs. one hop, respectively)

$$\text{Total\_Delay} = 2 \times (\text{D\_R\_ID}) + 2 \times (\text{A\_ID})$$

In synchronous designs, interconnect delay is the dominating contributor to total datapath delay. Doubling it, which four-phase asynchronous handshaking does, exacerbates the already big challenge of dealing with interconnect delays. To minimise the effect of this problem, an asynchronous (pipelined) routing switch was designed to be used as a side-switch. It is composed of multiplexers that route the inputs and request signals into a handshake-controlled latch (Figure 5.6). Using such switches breaks down long interconnect delays into smaller ones and hence reduces the effect of interconnect delay doubling.

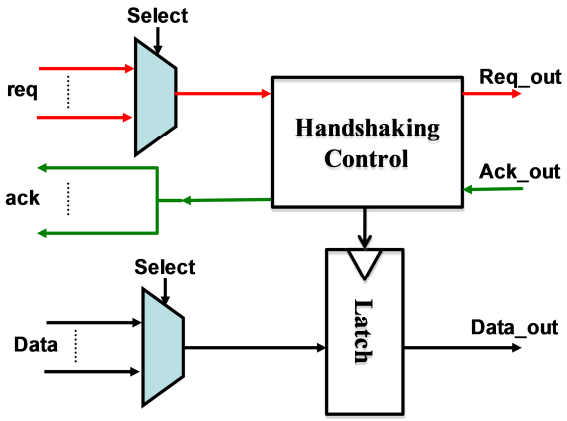


Figure 5.6: Asynchronous design of a routing switch.

The equation measuring the total interconnect delay in the pipelined interconnect case is the same as in the non-pipelined interconnect one (Equation 5.1). However, in this case, the D_R_ID and A_ID are broken down and fixed at a maximum of one switchbox delay as can be seen in Figure 5.7. The asynchronous routing switches can be regarded as another type of operational cell. For any connection between cells, the data-request interconnect delay is fixed at a delay of one routing multiplexer plus the reduced wire delay. Similarly, the acknowledge interconnect delay is fixed at a delay of one acknowledge synchronising block.

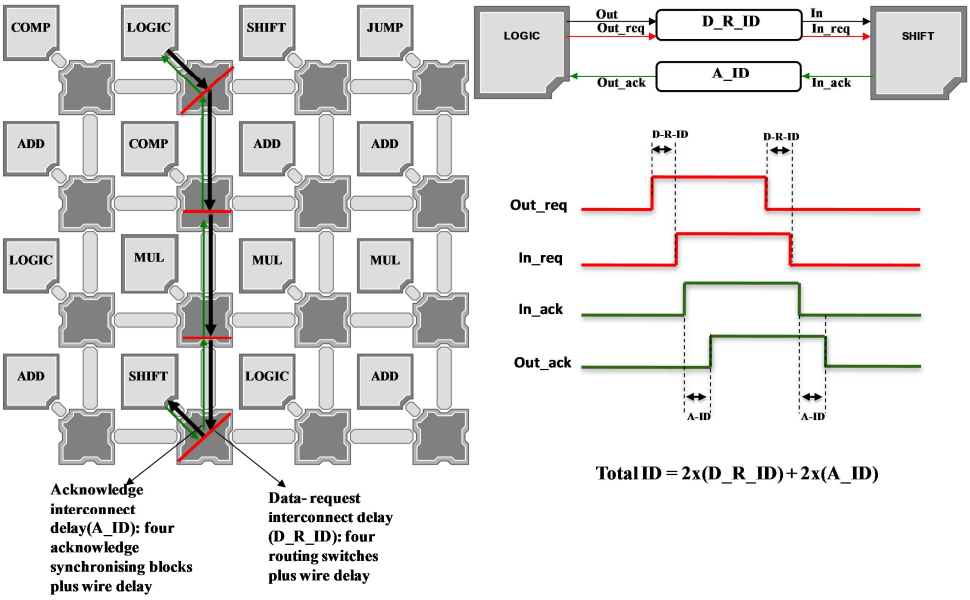


Figure 5.7: Interconnect delays in array using pipelined interconnects.

5.2.1 Multi-Channel Interconnect Design

An important consideration about the structure of the interconnect design is routability. It is defined as how likely the datapaths of a step are routable at the end of the tool flow. The routability of the interconnect networks greatly affects the gate utilisation and the speed of DRAP. Routability is related to the number of paths which are realisable in a given switch block array [98]. Rent's rule [99] also comes

into play: the larger the array size, the larger the datapaths that can be mapped. The larger the datapaths that can be mapped, the more interconnected the datapaths become. This increase in the average length of connections means that more interconnect resources (path segments) are used, which rapidly exhausts a single channel interconnect. It is therefore necessary to add more channels per switchbox in order to make the topology scalable.

The interconnect design in Figure 5.3 contains one input and output channel on each of its four sides and allows a maximum of 12 path combinations within it. To improve routability, the number of channels on the interconnect design were increased. A 5-channel switchbox was built. It is based on the design in Figure 5.3 but with five input and output channels on each side. Each output channel contains a routing switch and the size of the terminal-switch is larger to accommodate for additional signals to be routed into the asynchronous cell inputs.

A 5-channel switchbox allows a maximum of 60 path combinations within it. For the DRAP array, the 5-channel interconnect design was based on the Wilton switchbox [100] because it reduces area and delay while minimising the impact on routability.

5.2.2 Incorporating Cells in Routing Switch

Certain cells in DRAP such as the MUX and REG cells are required to be abundant and well distributed in the array. Muxing is used as an alternative to control flow, thus allowing larger kernels to be formed. Asynchronous register cells are mostly used for explicit pipelining. These cells are very small in area compared to the size of the switchbox. The area penalty of distributing these cells around the array, each with its own switchbox, is large. Additionally, it was found that muxing could be achieved through the existing interconnect muxes, thus avoiding the need for MUX cells. Since muxing is used a lot, this allows the cell and switchbox count to be reduced whilst achieving the same functionality.

As a result, these cells were incorporated into the DRAP interconnects. It is worth noting that this idea was first devised by the inventors of RICA (in July 2008). This was done by designing new routing switches. One routing switch, designed in Verilog, included a REG cell at its output. Another routing switch was designed in Haste. It can act as both an interconnect multiplexer or an asynchronous MUX cell depending on its configuration.

The designer, depending on the target applications, can choose the number of each cell within a switchbox. For the 15x15 array described in Section 8.2.2, a 5-channel switchbox with 2 REG cells and 2 MUX cells was used in the evaluations. This was adequate for the applications being evaluated.

5.3 Challenges of Interconnect Design for Asynchronous Reconfigurable Circuits

As mentioned before, the general interconnect structure of an asynchronous reconfigurable architecture can be conceptually modelled after that of an equivalent synchronous one. However, to communicate information, asynchronous circuits require more wires than their synchronous counterparts. Additionally, an asynchronous channel connecting a ‘send’ asynchronous block to multiple ‘receive’ blocks cannot be split or shared between the receivers without additional complex circuitry to acknowledge every transition on the channel.

In asynchronous logic, if a sender is connected to multiple receivers as shown in Figure 5.8, the request and data of the sender can be connected directly to all receivers. However, to ensure correct communication, the resulting acknowledge signals from all the receivers must be synchronised so that the sender receives an ‘acknowledge’ only when all the receivers have acknowledged its request. The gate labelled “C” is a standard multi-input C-element which synchronises the acknowledge signals. As shown in Section 2.1.4, the C-element works as follows:

when the inputs become equal, the output is made equal to the inputs; otherwise, if the inputs are not equal, the output is kept unchanged.

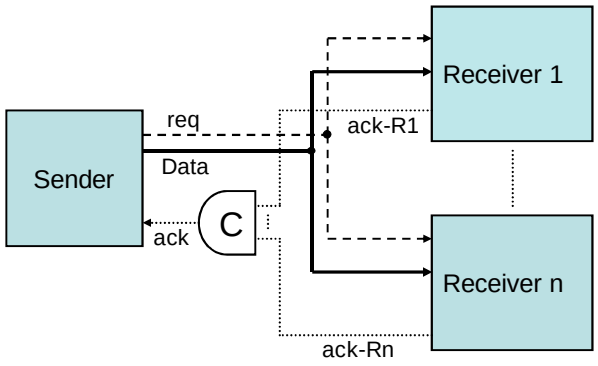


Figure 5.8: An asynchronous sender connecting to multiple receivers.

In programmable asynchronous logic, the sender will communicate with any number of receivers depending on what is being programmed. In this case, conditional synchronisation of the acknowledge signals must be performed.

An asynchronous reconfigurable architecture generally consists of operational cells, which can be fine-grain/coarse-grain and heterogeneous/homogeneous, and programmable switches for which different circuit design solutions and topologies exist [3]. Depending on the topology of the circuit, an operational cell would connect to other units via the programmable switch.

The DRAP architecture uses a topology where each operational cell communicates with its four neighbours. Depending on the programmed configuration, each operational cell can connect, through the programmable switch, to any number of combinations of the four other units.

The acknowledge signals from the receiving operational cells cannot be synchronised by simply using a tree of C-elements. Such a design requires all the receiving operational cells to acknowledge every request from the sending unit even if, say for a certain configuration, the sender is to communicate data with only one of its

neighbour's operational cells. The design of the programmable switch has to ensure that the handshaking between the operational cells always terminates for any configuration (Figure 5.9).

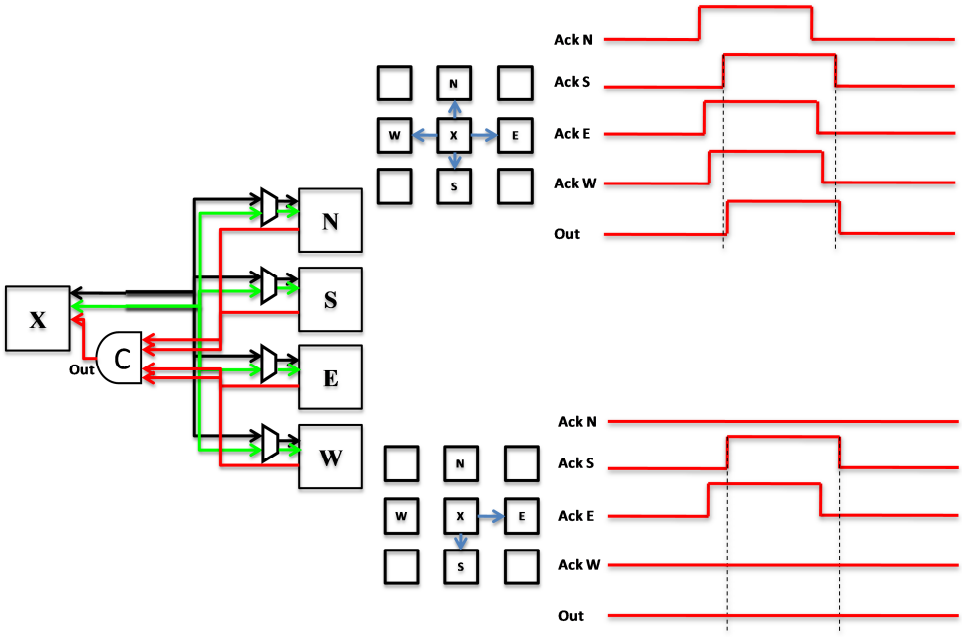


Figure 5.9: A cell (X) connected to its four neighbours using a C-element to synchronise the acknowledge signal: when the cell is programmed to connect to all four neighbouring cells (top configuration), the C-element synchronises all acknowledge signals after receiving them and correct communication occurs. When the cell is programmed to connect to only two neighbouring cells, E and S (bottom configuration), the C-element is waiting for an acknowledge signal from the N and W cells. These are not programmed to arrive hence communication between the cells is at a deadlock.

The next section explains how the conditional synchronisation of the acknowledge signals is done in other asynchronous architectures and presents the limitations of the techniques. In Section 5.5, a novel method which was designed and used in DRAP to perform conditional acknowledge synchronisation is presented.

5.4 Common Design Techniques

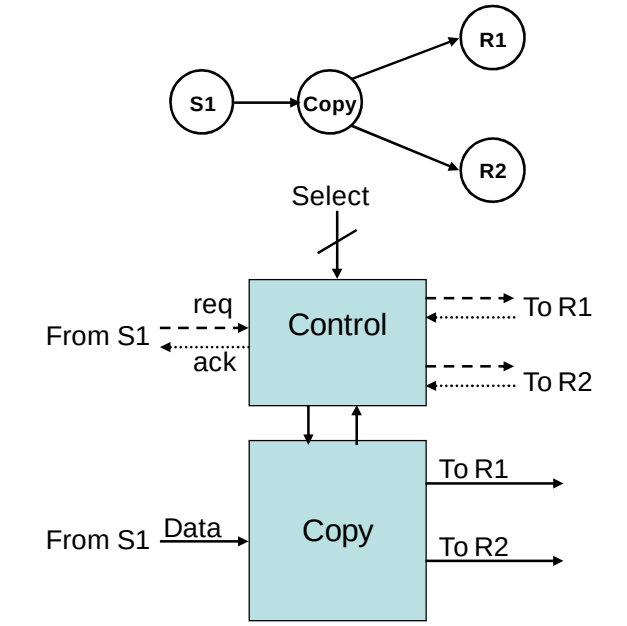
5.4.1 Overview

The technique used in available fully asynchronous reconfigurable architectures (such as in [86][87][90]) to build the interconnect structure is as follows: the cells and interconnect are designed so that all the tokens have unique senders and receivers. A token is defined as a group of inputs that are processed to produce a group of outputs. The term conjures up the notion of placing subway tokens on a circuit diagram and moving them around to visualise data moving through the circuit [101]. All channel routes should be made point-to-point and not fan-out to multiple receivers.

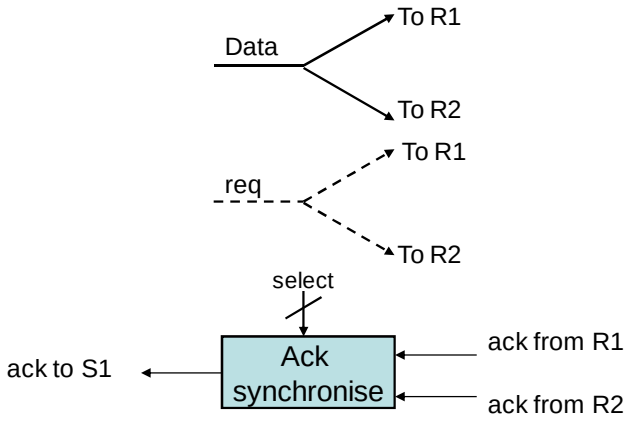
This means that tokens needed by more than one block must first be duplicated at a copy stage, where the request and data signals are duplicated and the multiple acknowledge signals are synchronised. A copy stage connecting a sender (S1) to two receivers (R1 and R2) must be designed to allow all possible duplication patterns to take place, i.e. copy a token from S1 to R1 or to R2 or to both.

Figure 5.10a shows a pipelined copy stage; the asynchronous control part is designed to implement handshaking at its inputs and outputs, to control the dataflow by controlling the latches and to perform conditional acknowledge synchronisation of the receiver acknowledge signals.

Figure 5.10b shows a non-pipelined copy stage. The request and data of a sender are sent to all receivers and the acknowledge signals are synchronised at a separate block. The figures in this chapter correspond to asynchronous bundled data encoding where the handshake signals and data are separate. All techniques discussed here also apply to N-of-M data encoding where the request signals are encoded within the data. Details on the different asynchronous data encoding techniques can be found in Section 2.1.5.



(a)



(b)

Figure 5.10: (a) Pipelined copy stage, which includes data latches controlled by the handshaking signals – the acknowledge signals are synchronised within the asynchronous control. (b) Non-pipelined copy stage where data and request are passed through and acknowledge signals are synchronised separately.

5.4.2 Techniques for Conditional Acknowledge Synchronisation

5.4.2.1 Traditional Technique for Conditional Acknowledge Synchronisation

This is the technique used in the interconnect design of the asynchronous reconfigurable architectures described in [86][90]. These architectures follow the island-style architecture used in typical FPGAs where each operational cell is connected to its four neighbours. Each operational cell has four inputs and outputs, equally distributed on its north, south, east and west sides. The output of the operational cell is duplicated to its four outputs through a pipelined copy stage. Figure 5.11 shows how the copy technique in [86] achieves conditional communication between the asynchronous blocks.

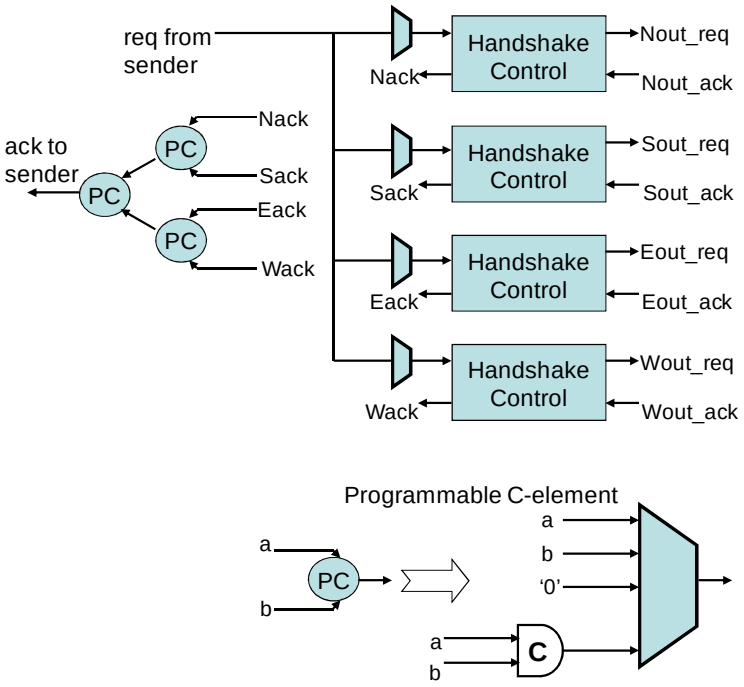


Figure 5.11: Pipelined copy stage used by the asynchronous reconfigurable architectures in [86]. Each cell connects to its four neighbours. A possible implementation of a programmable C-element is also shown (only control part of the circuit is shown).

The request signal from the sender is copied to the four destinations using four handshake controls. A multiplexer at the input of each handshake control allows the request signal to pass through, or not, depending on the configuration. The acknowledge signals from the handshake control are combined using a tree of programmable C-elements and other configurable logic that allow the inputs of a C-element to be ignored or asserted depending on the configuration. The resulting copy stage is large and requires six configuration bits for synchronising the acknowledge signals in addition to the configurations bits needed to route the request and data signals.

The method traditionally used for performing the conditional acknowledge synchronisation part of the copy stage is to use a tree of C-elements and other configurable logic that allow the inputs of a C-element to be ignored or asserted depending on the configuration. The advantage of this method is that it does not place any condition on the topology of the device. The disadvantages of this method are as follows:

- Increase in the number of configuration bits: This is in comparison to an equivalent interconnect design for synchronous communication. Additional configuration bits are required to control the conditional synchronisation of the acknowledge signals through controlling C-element tree structure.
- Increase in pipelined copy stage complexity: Each of the output channels requires its own basic handshake channel with a C-element for each. The output of the C-element on each channel is routed through more control logic which depending on the value of the select signal, returns an ‘acknowledge’ to both sides of the handshake channel.

5.4.2.2 Reduced Complexity Technique for Conditional Acknowledge Synchronisation

This is the technique used in the asynchronous reconfigurable architecture described in [87]. Each operational cell is made up of a collection of four 3-input logic ‘control blocks’. Each ‘control block’ is limited to connecting to a maximum of two ‘control blocks’ in another operational cell. The request signal from the sending ‘control block’ is routed to both receiving ‘control blocks’ via a handshake control - a circuit that implements four-phase or two-phase handshakes on its inputs and outputs and controls the data latches [13].

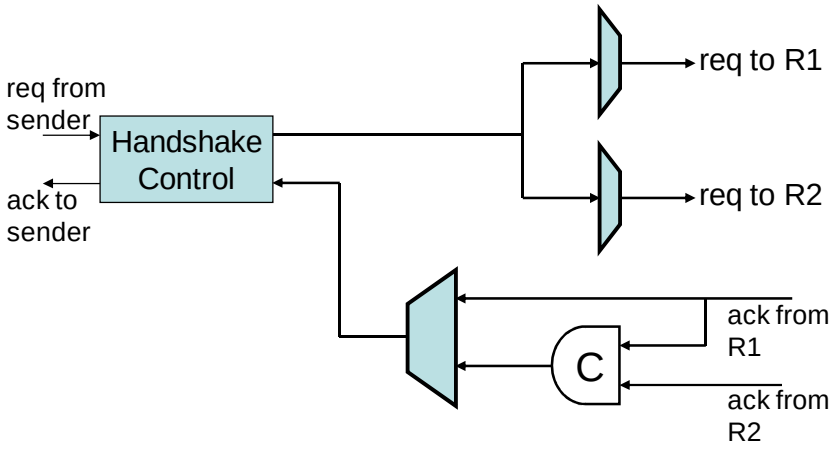


Figure 5.12: Pipelined copy stage used by asynchronous reconfigurable architecture in [87]. Each cell is limited to connect to two other cells (only control part is shown).

A multiplexer selects whether, depending on the configuration, the acknowledge signal comes from one or both of the receiving ‘control blocks’ (Figure 5.12). Compared to the traditional technique of Section 5.4.2.1, the complexity of the conditional acknowledge synchronisation of the copy stage design is reduced. This technique however sacrifices flexibility of the device for simplicity in designing the conditional acknowledge synchronisation. Additionally, each ‘control block’ in the operational cell requires an extra bit to choose the correct acknowledge signal on top of the configuration bits needed to route the request and data signals.

5.5 Proposed Technique for Acknowledge Synchronisation

A new method (Figure 5.13) for performing conditional communications in programmable asynchronous logic is presented here. The proposed technique minimises control and configuration size compared to existing techniques. The scheme employs select signals that are already used to control the data routing switches, to control the synchronisation of the acknowledge signals. The select signals identify the active acknowledge signals, which are then routed through. The inactive acknowledge signals, where no data will pass through, are preset. The data and request signals of a sender are fanned-out to all its receiving routing switches.

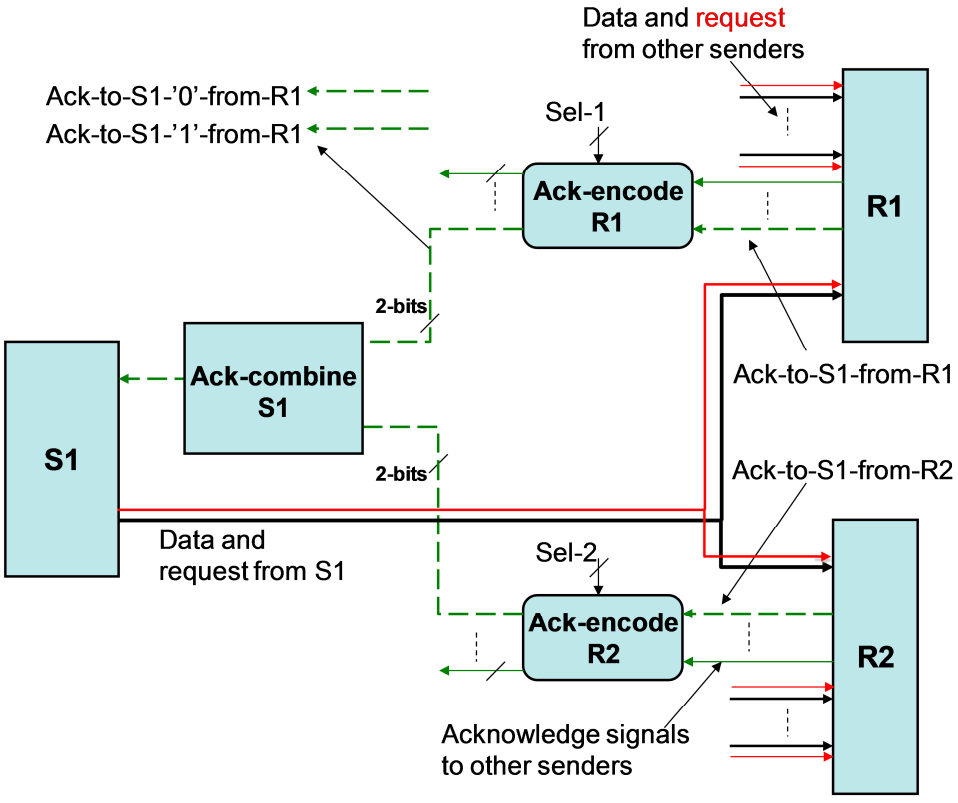
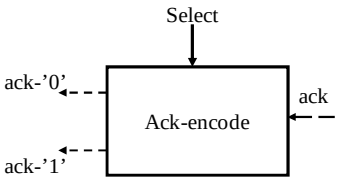


Figure 5.13: Overview of the acknowledge synchronising method.

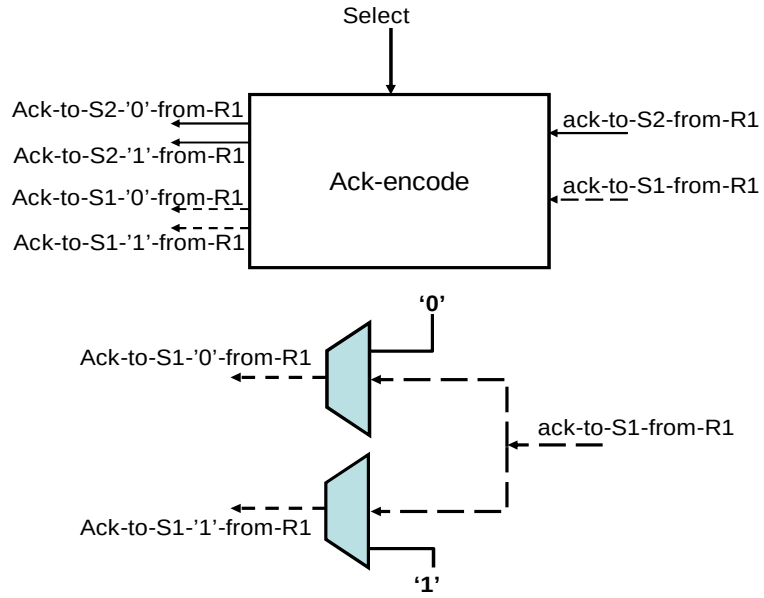
The resulting acknowledge signals are fed at the receiver stage into an ack-encode block, which depending on its select signal (i.e. depending on whether the acknowledge signal is active), encodes the acknowledge signals into 2-bit signals (Figure 5.14). The encoded signals are fed back to an ack-combine block at the sender stage, which combines all the 2-bit acknowledge signals of that sender. Compared to the conditional acknowledge synchronisation techniques described in Section 5.4.2, this technique does not require any additional configuration bits, since the same select signal of the routing switch is fed into the ack-encode block. Additionally, no extra handshaking control circuit is required for every route from a sender to a receiver and hence complexity is reduced.



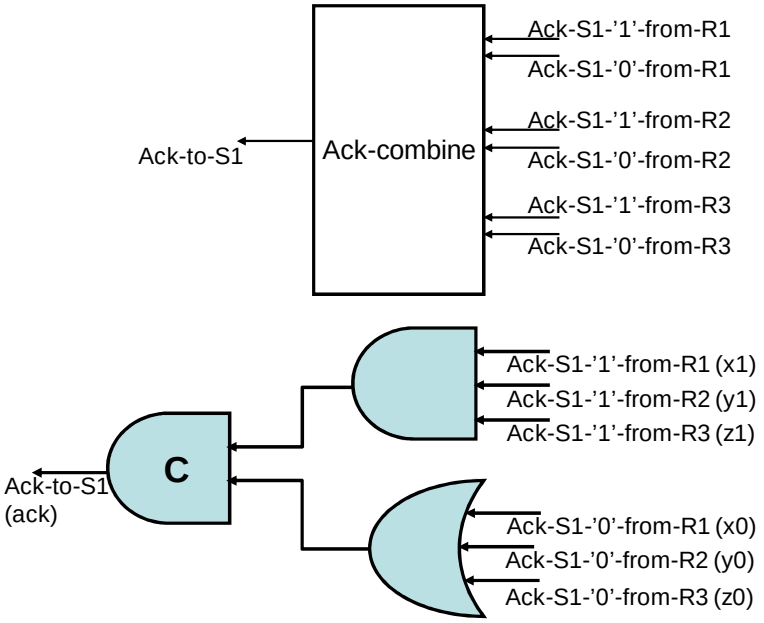
ack	Select (active)	ack-'0'	ack-'1'
0	Yes	0	0
1		1	1
0	No	0	1
1		0	1

Figure 5.14: Ack-encode truth table. The select signals identifies if the ack signal is active or not. If it is, then the ack signal is routed through to ack-'0' and ack-'1'. Otherwise, ack-'0' is set to logic 0 and ack-'1' is set to logic 1.

The ack-encode block (Figure 5.15a) takes in as inputs the acknowledge signals from a receiver (each belongs to a particular sender) and a select signal from the configuration, and outputs two acknowledge signals per input. If an acknowledge signal (ack-to-S) is selected by the configuration, its logic value is copied to its corresponding two output signals (ack-to-S-'0' and ack-to-S-'1'). The ack-to-S-'1' and the ack-to-S-'0' of all other unselected inputs are set to logic high and low respectively.



(a)



(b)

Figure 5.15: (a) Ack-encode of a receiver (R1), which is connected to two senders (S1, S2). (b) Ack-combine of a sender (S1) which connects to three receivers (R1, R2, R3).

The ack-combine block (Figure 5.15b) of a sender receives all the encoded acknowledge signals from all the receivers which it is connected to. The resulting combined acknowledge signal switches from high/low to low/high only when all the receivers connected to this sender have acknowledged the request.

Figure 5.15b shows an ack-combine of a sender (S1) which connects to three receivers (R1, R2, R3). Assume for a configuration that S1 is connected to R1 and R3; y1 and y0 would be set by the R2 ack-encode to high and low respectively and x1/x0 and z1/z0 would be connected to the acknowledge signal from R1 (x) and R3 (z) respectively. The AND gate and C-element ensure that acknowledge goes high only after x and z both go high. When the S1 request goes low and consequently x and z go low, the OR gate and C-element ensure that acknowledge goes low only after both have become low.

Figure 5.16 shows an abstract overview of how the proposed technique fits around an operational cell.

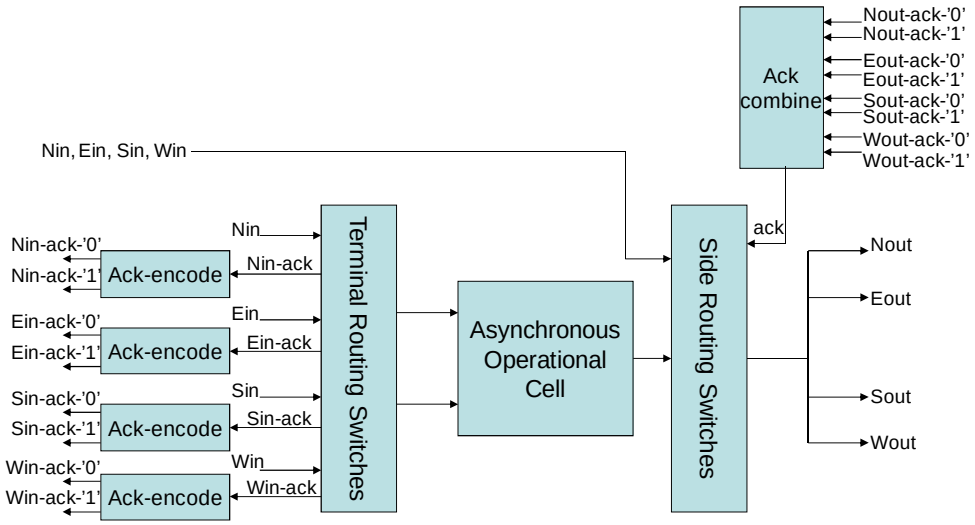


Figure 5.16: Block diagram of an operational cell and switchbox using the proposed technique to synchronise acknowledge signals.

5.6 An Interconnect Design Comparison

To evaluate the proposed technique, five variations of the DRAP switchboxes were designed. The switchboxes were single channelled with one input and output on each of their four sides as shown in Figure 5.3. A non-pipelined routing switch (Figure 5.4) was used for all the terminal-switches. Two switchboxes which used the proposed technique for synchronising the acknowledge signals were designed, one with pipelined (Figure 5.6) and one with non-pipelined routing switch for the side-switches. Another two switchboxes which use the traditionally used method for synchronising the acknowledge signals were built. Both use non-pipelined routing switches as side-switches but one used a pipelined copy stage similar to design in Figure 5.11 and the other a non-pipelined one (Figure 5.10). The pipelined routing switch and pipelined copy stage were designed using the TiDE tool.

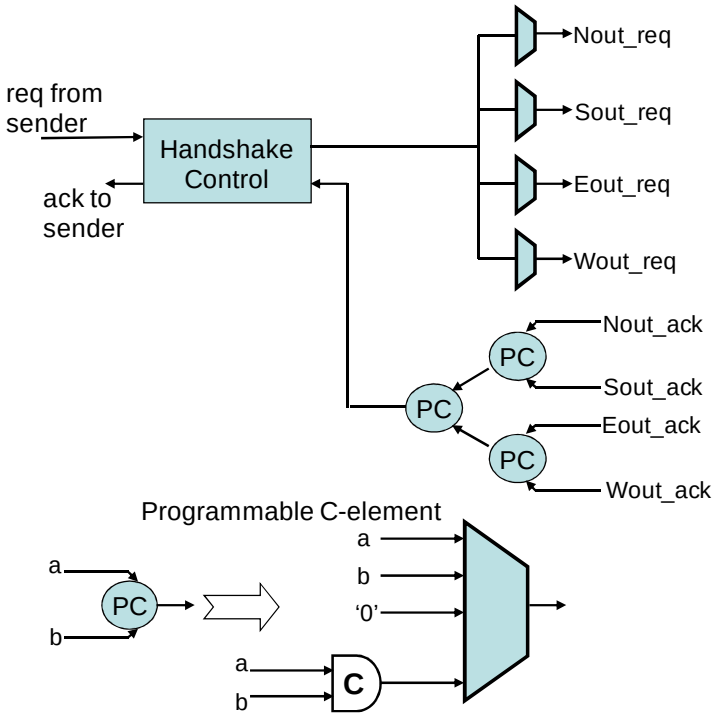


Figure 5.17: An optimisation applied to copy stage design of Figure 5.11 (The Ack-encode are at the receiver side).

An optimisation to the design of the pipelined copy stage shown in Figure 5.11 was identified. A reduction in control area was achieved by reducing the number of required handshaking control blocks through moving the parts which select the request signals and synchronise the acknowledge signals to the output of the copy stage. This is referred to as an optimised copy stage (Figure 5.17).

To summarise, the following switchboxes were built and compared:

- Non-pipelined using proposed technique (to synchronise the acknowledge signals).
- Non-pipelined using 'traditional copy technique'.
- Pipelined using proposed technique.
- Pipelined using 'traditional copy technique'.
- Pipelined using optimised copy technique.

Five 16-cell arrays (4x4), with an identical distribution of cells and each using one of the types of switchboxes described above were generated with UMC0.13 technology. A sample radix-2 FFT algorithm [102] was mapped on each array. The area and power consumption for the designs are measured at post-layout level (using Synopsys and Cadence tools). The presented power values are the average of power consumed by all the switchboxes for each array. All the power data are measured at 1.2-V operating voltage using Synopsis PrimeTime PX. The power measurements are for total power (glitch power included), which includes both active and leakage power. More details of the FFT implementation can be found in Section 8.1.2. The results are listed in Table 5.2 through Table 5.4 and Figure 5.18.

Table 5.2: Comparison of the number of configuration bits, normalised gate area and power consumption of the switchboxes designed both the proposed technique and the traditionally used technique.

Design	Area	No of config bits/switchbox	Power
Proposed, non-pipelined	1	18	1
Traditional, non-pipelined	1.20	24	1.10
Proposed, pipelined	1.23	18	1.15
Traditional, pipelined	2.35	24	2.10
Optimised traditional, pipelined	1.5	24	1.3

Table 5.3: Minimum Interconnect delay introduced by using the different acknowledge synchronising methods in a non-pipelined switchbox. The delay is based on connecting the output of a cell to the input of its neighbouring cell.

Non-pipelined switchbox	Data-request ID (ns)	Acknowledge ID (ns)	Total Delay for 4-phase handshaking (ns)
Proposed technique	0.45	0.9	2.7
Traditional technique	0.45	1.2	3.3

Table 5.4: Fixed interconnect delay introduced by using the different acknowledge synchronising methods in a pipelined switchbox.

Pipelined switchbox	Data-request ID (ns)	Acknowledge ID (ns)	Total Delay for 4-phase handshaking (ns)
Proposed technique	0.23	0.62	1.7
Traditional technique	0.35	0.67	2.0
Optimised traditional	0.35	0.67	2.0

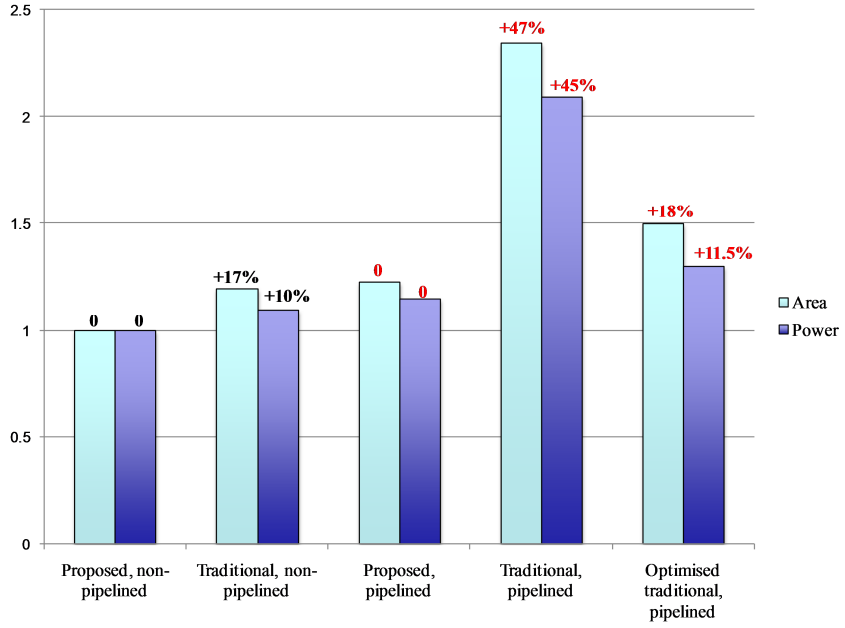


Figure 5.18: Normalised area and power consumption of the designs.

5.6.1 Proposed vs. Traditional Methods

Table 5.2 shows that the proposed acknowledge synchronising technique leads to the design of switchboxes which require 25% less configuration bits (18 rather than 24 bits) than ones using the traditional copy technique. Furthermore, the non-pipelined and pipelined switchboxes using the proposed technique are 17% and 47% smaller in area and consume 10% and 45% less power than their equivalent implementations using the traditional technique. Compared to the optimised traditional technique, the pipelined switchbox using the proposed pipelined technique consumes 11.5% less power and takes up 18% less area.

Table 5.3 through Table 5.4 show that, compared to the traditional technique, the proposed technique introduces 18% and 15% less delay in non-pipelined and pipelined switchboxes respectively.

5.6.2 Pipelined vs. Non-pipelined Switchboxes

From Table 5.2 through Table 5.4, a comparison between using pipelined and non-pipelined switchboxes can be drawn. Obviously, pipelined switchboxes consume more area than non-pipelined ones due to them containing latches and their corresponding handshake control. Pipelined switchboxes using the proposed technique take up 23% more area and consume 15% more power than non-pipelined ones.

However, pipelined switchboxes limit the delay introduced by the interconnects at a maximum of 1.7 ns (total delay). In comparison, the delay introduced by non-pipelined switchboxes is variable and depends on the number of switchboxes connecting two operational cells. As can be seen in Figure 5.19, the minimum delay, achieved by a one-switchbox connection, is 2.7 ns. This number rises linearly in a rapid way as the number of switchboxes connecting two cells increases.

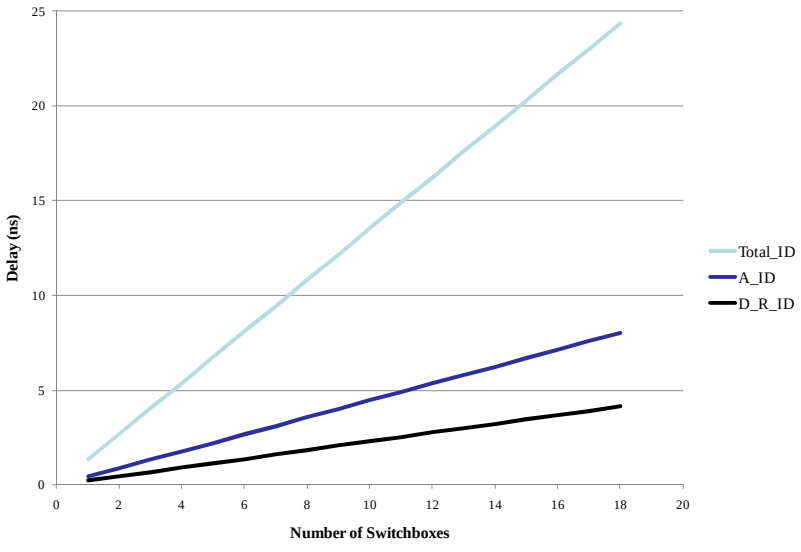


Figure 5.19: Plot showing how the total interconnect delay (Total_ID), acknowledge interconnect delay (A_ID), and data-request interconnect delay (D_R_ID) vary with the number of switchboxes connecting two cells.

5.7 Summary

This chapter described the second part of the design of the DRAP hardware: the interconnect structure, its variations, and how it was built. The interconnect structure for DRAP was based on the island-style design used in FPGAs. Pipelined, non-pipelined, and multi-channel switchboxes were designed. The second half of the chapter explained the main challenges in designing interconnects for asynchronous reconfigurable circuits and the techniques used by other asynchronous reconfigurable architectures to address them. These techniques were shown to result in large and complex interconnect designs which needed more configuration bits than an equivalent interconnect for a synchronous architecture. Finally, a novel method for designing interconnects for asynchronous reconfigurable architectures was presented and compared to ones found in the literature. The new method resulted in interconnects which were up to 47% smaller in area, consumed up to 45% less power and required 25% less configuration bits than interconnects designed using the traditional techniques.

Chapter 6

Controlling Reconfigurability in DRAP

The DRAP and RICA arrays allow their operational cells to connect and create datapaths of varying lengths. Hence the critical path of each configuration context changes depending on the datapaths being mapped and the level of pipelining required. To account for this, most programmable devices limit the maximum operating frequency to the largest critical path delay of all the steps (configuration contexts) being mapped.

As shown in Section 2.5, RICA uses a Reconfiguration Rate Controller (RRC) to control the length of time for which each step persists. For each configuration context, the critical path delay is estimated in software, and the resulting divisor is programmed as part of it. The RRC connects to all the sequential cells such as registers (synchronous cells) in RICA. When the RRC expires, the state of the synchronous cells is updated, and the configuration context referenced by the program counter, which is managed by the JUMP cell, is loaded. The program counter may refer to the same step that just ended, which is the case for kernel steps. In such cases, the configuration context persists for another iteration, without

incurring any transactions from program memory, or any reconfiguration delay. This is the most efficient way to execute kernels on RICA.

DRAP, on the other hand, has no global clock and the cells communicate via handshaking which implements synchronisation among the components of the asynchronous datapath irrespective of its length. A benefit of this is that it eliminates the need for the RRC and for delay estimation in software and the time quantisation it infers. However, a method is needed to indicate when the datapaths in a configuration context can start computing and when they have finished and the next step can be loaded.

An Asynchronous Reconfiguration Controller (ARC) was designed to indicate when a step has concluded its work. It extracts a FINISH signal from the handshaking signals of certain cells and feeds it to the JUMP cell. The JUMP cell acts as the instruction-controller and manages the program counter. It also indicates when the datapaths in a configuration context can start processing data. This chapter explains how DRAP controls its reconfiguration. The designs of the ARC and the JUMP cell are explained and the method is compared to that used in RICA.

6.1 Overview

Figure 6.1 shows how the ARC and the JUMP cell interact with the other cells in the array. EP is the label given to all the cells which could form the endpoint of a datapath. These constitute all the asynchronous register cells and the SBUF and SINK Interface cells (see Section 4.4). The endpoint cells are connected directly to the ARC via a DONE signal which indicates that the cell has finished its computation. For each configuration context, the ARC monitors only the signals of the endpoint cells which are in use in that step.

When all the chosen signals go high, the step has finished computing and the FINISH signal from the ARC to the JUMP cell goes high. The JUMP cell in turn updates the program counter and the corresponding configuration context is loaded. The JUMP

cell also raises a signal labeled Start_to_delay. This signal is fed to all the cells that maintain state in the array (the asynchronous register cells, the line buffer cells and the JUMP cell – see Section 4.4). The signal indicates to the cells that the configuration has been loaded and it is safe to start the computational process. This is why the Start_to_delay signal passes through a delay block which matches the time the program memory needs to load onto the array.

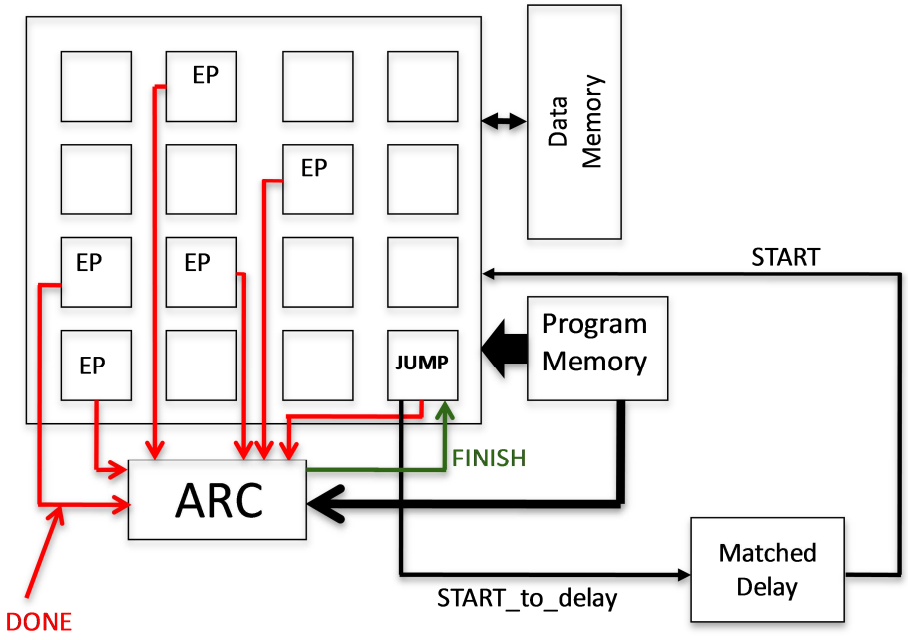


Figure 6.1: Overview of how reconfiguration is controlled in DRAP. The Asynchronous Reconfiguration Controller (ARC) interacts with certain cells in the array - the JUMP and endpoint (EP) cells - to indicate when a step has finished its work.

6.2 ARC module design

The function of the ARC module is to indicate when a step has finished computing and hence the next configuration context can be loaded. The ARC receives DONE signals from all possible endpoints and extracts a FINISH signal. There are two types of endpoints, the asynchronous register cells and the line buffer cells. The DONE

signals from each type of endpoints are treated separately within the ARC (see Figure 6.2).

6.2.1 Extracting the FINISH signal

The DONE signal from an asynchronous register cell is derived from the acknowledge signal of the register input. Each signal is fed to a negative edge detector circuit in the ARC. When a change from high to low is detected - i.e. the cell has finished the fourth and last stage of its handshake (Section 2.1.3) - the edge detector output goes high.

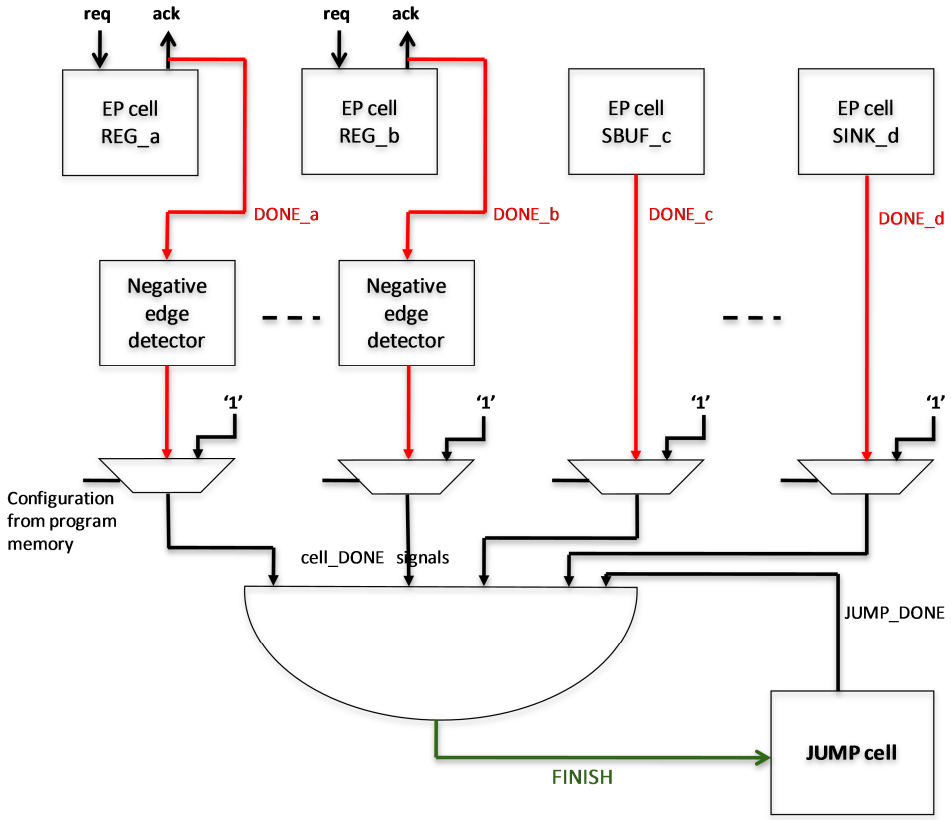


Figure 6.2: A closer look at how the Asynchronous Reconfiguration Controller (ARC) interacts with the CJUMP and endpoint (EP) cells.

As for the line buffer cells, the DONE signal is taken directly from each cell. Each Line buffer cell contains an asynchronous counter that is controlled by the program memory. When the programmed number of data has passed through the cell, a signal indicating it has finished goes high; that signal is the DONE signal.

The output of the edge detector for each register cell, and the DONE signal taken directly from each line buffer cell, are fed through a multiplexer controlled by the program memory. Depending on the loaded configuration context, the multiplexer allows the signal to pass through if it is an endpoint cell in that step or outputs a logic high. The output of the multiplexer is known as the cell_DONE signal.

The JUMP cell can also be an endpoint and might in some steps be the last cell in the critical path. That is why the JUMP cell is designed to output a JUMP_DONE signal which indicates it has finished its task. The JUMP_DONE signal and the cell_DONE signals from the register and line buffer cells are combined via a tree of AND gates to derive the FINISH signal (see Figure 6.2). There is a scalability problem with this approach - the AND tree gets slower as the core gets larger, quickly dominating the critical path delay.

6.2.2 Improvements

Certain improvements to the above technique of extracting the FINISH signal (see Figure 6.3) were identified and implemented, in order to reduce the effect of the AND-tree delay, and thus improve scalability.

The advantage of the above approach is that it places no conditions on the software part of the design (routing).The main disadvantage however is that it requires all asynchronous register cells to be endpoint cells. As their number increases in an array, the hardware required to extract the finish signal becomes larger and slower. A more balanced approach was needed which would divide the task of extracting the FINISH signal between hardware and software and result in a scalable technique.

The first improvement was thus to reduce the number of endpoint signals. Instead of designating all the asynchronous register cells as endpoint cells, the number was limited to just a few and does not have to increase linearly as the array gets bigger. The choice of the number of endpoint cells could be chosen by the designer depending on the family of applications. The mapping tools are now required to ensure the critical path of each step ends with one of these designated endpoint cells. This is discussed further in Section 0 and Section 7.2.

The main kernels of DRAP's targeted applications (mobile applications) typically end with a SINK cell external interface cell (Section 4.4) – it lies at the end of the critical path. As a second improvement, the SINK cell DONE signals were separated from DONE signals of the other endpoint cells (see Figure 6.3). This results in a reduction of the AND-tree delay in both kernel and non-kernel steps.

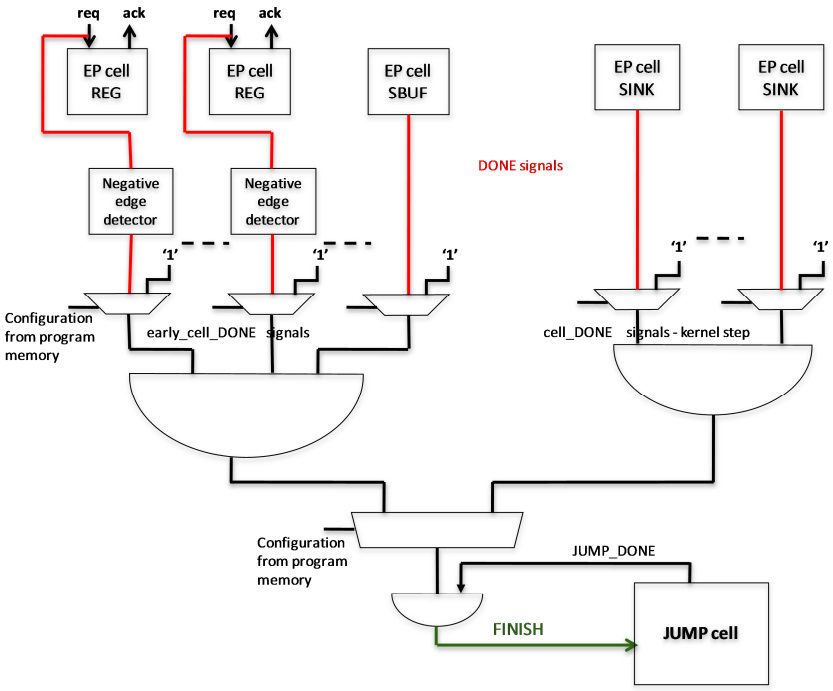


Figure 6.3: A closer look at the optimised Asynchronous Reconfiguration Controller (ARC) and its environment. The AND gate delay is reduced and the ARC uses the early_cell_DONE from the endpoint (EP) cells.

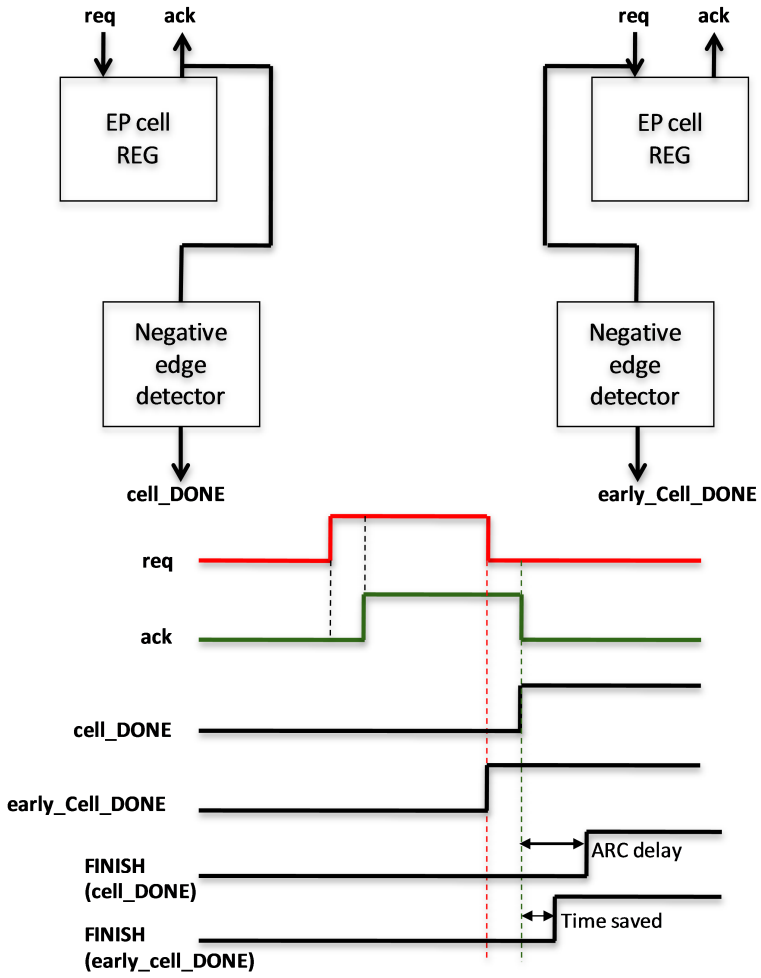


Figure 6.4: Extracting cell_DONE and early_cell_DONE signals for the endpoint (EP) cells.

The final improvement involved extracting an early_cell_DONE signal from the asynchronous register cells in order to reduce the effect of the ARC delay (a combination of the edge detector, AND-tree, and multiplexer delays). Choosing the DONE signal of a register cell from its input request signal instead of its input acknowledge signal is a way of achieving this. The negative edge detector would see a transition from high to low at the third handshake phase. As long as the time needed to terminate the handshake (from third to fourth phase) is less than the ARC delay, this method reduces the effect of the ARC delay on extracting the FINISH signal (see Figure 6.4).

6.3 Scalability of technique

As discussed in Chapters 1 and 3, one of the benefits of DRAP, associated with its asynchronous nature, was increased scalability. The target of the DRAP architecture is mobile applications. Since most of these applications use large kernels, it is desirable to increase the number of operational cells in the array in order to run such applications at higher throughputs by mapping the entire kernel onto one configuration context. When a RICA core gets bigger, the number of sequential elements (primarily registers) in the array must be increased significantly in order to maintain routability and pipelining. As a result, the clock tree area increases and it consumes more power. By using asynchronous design techniques, DRAP eliminates the clock tree and replaces it with local handshaking which implements synchronisation among the operational cells of an asynchronous datapath irrespective of its length. However, a method to control how much time a configuration context must persist for is still required. It is important that the devised method of controlling reconfigurability in DRAP maintains the benefit of scalability that asynchronous reconfigurable architectures have over synchronous ones.

A novel technique developed to control reconfigurability in DRAP has been described. It depends on designating certain cells in the array as endpoint cells i.e. cells where the critical datapath of a configuration context has to end at. There are two classes of cells that can be endpoints, the asynchronous register and the write interface cells (SBUF and SINK). Allowing all those cells in an array to be designated as endpoint cells simplifies the routing software but adds complexity and delay to the ARC hardware. This is because of the large number of multiplexers and their corresponding configuration bits as well as the large tree of AND gates needed to extract the FINISH signal. As the array gets larger, more endpoint cells are needed. Hence, the ARC becomes slower and more complex and DRAP loses its advantage of scalability.

Section 6.2.2 presented a hardware solution to improve the scalability of the ARC technique, by limiting the number of designated endpoint cells. By doing this, the ARC hardware is simplified and the AND tree delay is greatly reduced. However, there is now an additional constraint on the routing part of the DRAP tool flow – to make sure the critical path of a configuration step terminates at an endpoint. The DRAP tool flow is based on those developed for the hardware generation and programming of RICA (see Section 7.2). Figure 6.5 best describes the constraint and the methods used to address it. Assume one of the steps of a complied program has four datapaths where datapath 2 is the critical path. It is possible that after the routing stage of the tool flow, the critical path now becomes datapath 3.

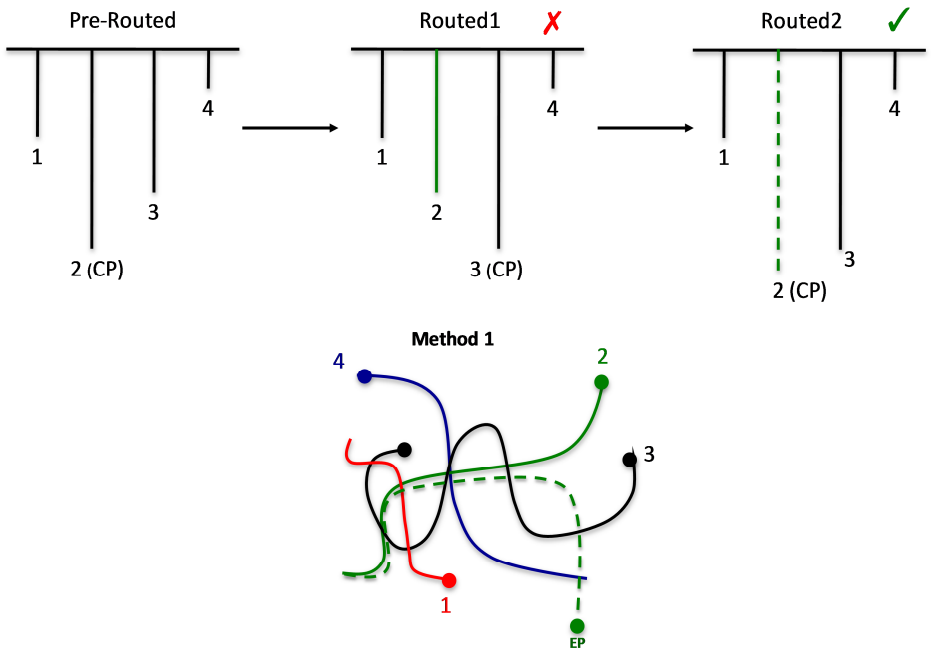


Figure 6.5: An example of a configuration with four data-paths. The routing algorithm is required to ensure that the pre-routing critical path remains the critical path after routing and ends in one of the end-point cells. This adds significant pressure to the routing algorithm.

The first solution (method 1) is to force datapath 2 to be routed as a critical path and terminate in one of the possible endpoints. This solution requires the complicated and difficult step of changing the routing algorithm to ensure always that the critical path

before routing is still the critical path after routing and that it terminates in an endpoint cell. Additionally, this solution depends on the number of endpoint cells present in the array. As the array increases in size, the number of endpoint cells must be increased to avoid the routing becoming over-constrained. As the number of endpoint cells increases, the delay of the AND tree also increases. Both of these factors again reduce the scalability of the architecture.

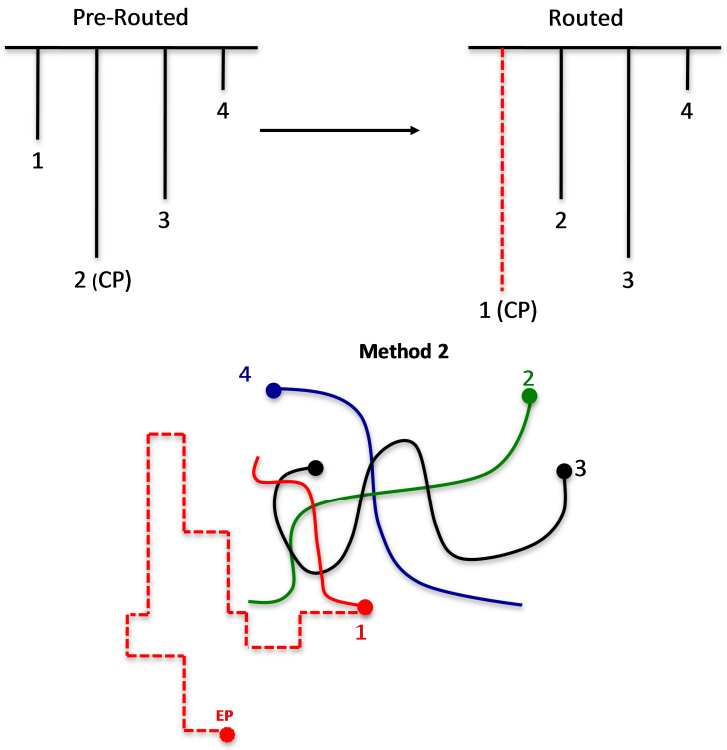


Figure 6.6: The routing algorithm simply chooses any of the datapaths in a configuration context and artificially extends it to become the critical path and terminate at an endpoint cell. With this method, the additional pressure on the routing algorithm is removed and both hardware and software are simplified. Additionally this method is scalable as the number of endpoint cells need not increase as the array increases.

An alternative solution (method 2) is to adjust the routing part of the tool flow to calculate the price of artificially extending each datapath in a configuration context to become the critical path and terminate at an endpoint cell. The best choice

datapath is selected; this is defined as the datapath which requires the shortest extension to be turned into an endpoint-terminating critical path.

With this method (Figure 6.6), the existing routing algorithm was reused and both hardware and software simplified. Additionally this method is scalable as the number of endpoint cells need not increase as the array increases. For example, nine endpoint cells were used in two tested DRAP designs: a 20x20 array and a 15x15 array. Extending a datapath would add delay in a non-kernel step but will not affect kernel steps, due to being hidden by pipelining. And since kernels take up over 95% of execution time, the effect of the added delay is negligible.

In addition to being scalable in terms of power and area, the method of controlling reconfigurability in DRAP is also hardware timing independent. If the array is fabricated on the next process node down, the cell delays should all be affected in more or less the same way. This means that the critical path is the same cell or number of cells, but a shorter time. The ARC only cares about getting signals from the endpoint of the critical path and not what the length of the critical path is. However in synchronous RICA, the length (delay) of the critical path is important and affects the configuration value, which may not scale when fabricated at a different size.

6.4 Jump cell

Like RICA's JUMP cell, the asynchronous JUMP cell (Figure 6.7) in DRAP manages program control flow. The JUMP cell contains the program counter, which specifies the index of the next step to be executed. If this differs from the step currently executing, the new index is sent to the program memory controller to retrieve the configuration of each cell in the new step. Additionally the JUMP cell controls the register (REG) and interface (SOURCE, SINK, SBUF) cells by indicating to them when it is safe to start computing.

During the execution of a step, the JUMP cell computes the next value of the program counter so that the configuration of the next step would be ready when needed. The computation of the next location can be conditional by using the output of another cell (e.g. ADDCOMP), and hence achieving conditional branching in DRAP.

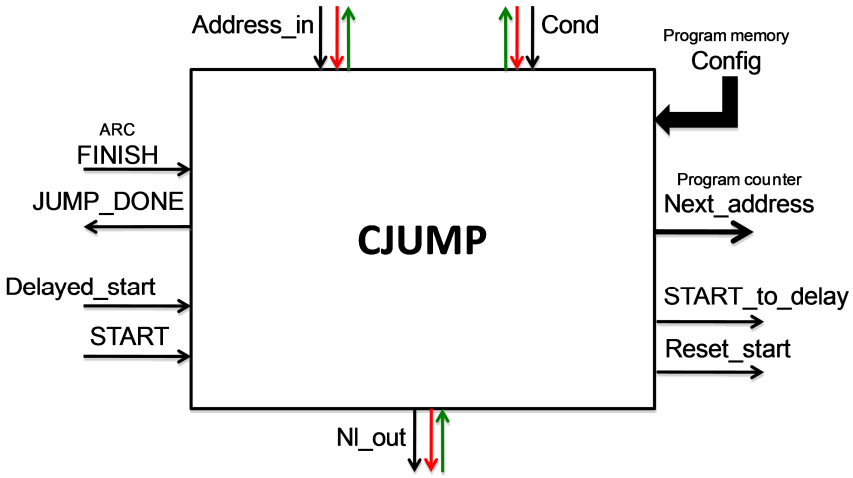


Figure 6.7: Block diagram of the asynchronous CJUMP cell.

Figure 6.8 shows how the CJUMP interacts with its environment. It calculates the next program memory address (next_address) based on its two data inputs and the current address. This varies depending on whether the step is a kernel or not. CJUMP raises the JUMP_DONE signal when it has finished the calculation and waits for the FINISH signal to go high and hence indicate it is now safe to load the next step.

If the next step is going to be a kernel, then the CJUMP raises the START_to_delay signal and keeps it high for the duration of the kernel step. The Delayed_start and Reset_start signals are used to promptly lower the START signal. Without them, the START signal could be kept high for longer than needed and cells which receive this signal will continue to output data and wait for their handshakes to be acknowledged when they should not.

For kernel steps, the mux is programmed to select one of the kernel_finish signals - the one corresponding to the designated endpoint cell for that step. This way, in kernel steps, the JUMP receives a FINISH signal only when the kernel has terminated and updates the program counter only once at the end of the kernel to jump to the next step. If the next step is a non-kernel step, CJUMP sends a pulse down the START_to_delay signal. This allows the start receiving cells to issue one output only.

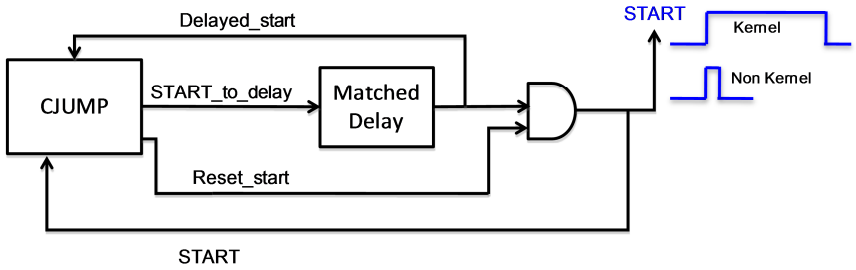


Figure 6.8: Overview of how the CJUMP cell interacts with its environment.

6.5 Summary

The hardware-software mechanism of how the DRAP array controls its reconfiguration was described in this chapter. For the hardware part, the ARC indicates when a step has concluded its work by deriving a FINISH signal from the handshaking signals of certain cells called the endpoint cells and feeding it to the JUMP cell. The JUMP cell acts as the instruction-controller and manages the program counter. It also indicates when the datapaths in a configuration context can start processing data.

The designs of the ARC and the JUMP cell were explained in more detail. For the software part, the tool programs the ARC and JUMP cells. It also ensures during the routing stage that for each step, the datapath representing the critical path terminates in an endpoint cell. The solution presented in this chapter was scalable in terms of area and power.

Chapter 7

Automatic Tool Flow

An automatic tool flow has been developed for the generation of DRAP arrays. The DRAP tool flow is based on the tools for hardware generation and programming of the RICA synchronous coarse-grain arrays. These were adapted to include support for asynchronous cells and interconnects, and for controlling reconfigurability in the asynchronous array (ARC). This chapter begins by defining implicit and explicit pipelining. The definitions are important to understand why and how some parts of the original tools were adjusted. The next part of the chapter describes the different parts of the DRAP tool flow.

7.1 Implicit and Explicit Pipelining

Reconfigurable datapath architectures like RICA and DRAP allow the kernel of many streaming applications to be mapped entirely in a single step. This step persists for as many iterations as needed to operate on a batch of data. The large number of operations in such kernels often leads to long critical paths, which limit the

throughput. A common method of increasing the throughput is to pipeline the datapaths in the kernel, so as to reduce the effective critical path of an iteration, thus increasing the iteration rate. Pipelining allows multiple iterations to be in-flight within the core at once. On DRAP, there are two ways in which to do this - explicit pipelining and implicit pipelining.

7.1.1 Explicit Pipelining

When forming kernels, it is possible to partition the operations into pipeline stages, and insert registers between connected operations that lie in different pipeline stages. Register cells in the core (or interconnect) are used for this purpose. This technique is called explicit pipelining, and was introduced in RICA [4][68] as a way of significantly increasing the throughput. However, inserting registers into the appropriate datapaths requires routing those datapaths through registers in the core. This increases routing pressure, and requires that additional registers be available in the core in order to maintain the same level of routability.

7.1.2 Implicit Pipelining

Asynchronous cells contain latches at their inputs and/or output. These latches are controlled locally and automatically by the cell's handshaking control unit. The availability of the built-in latches provides full pipelining within many datapaths. This property of asynchronous cells to provide inherent pipelining of datapaths is referred to as implicit pipelining. Implicit pipelining requires no additional registers (the datapaths are unmodified), and thus introduces no additional routing pressure.

7.1.3 Explicit vs. Implicit Pipelining

Explicit pipelining on RICA requires connecting additional registers into the datapaths, which increases demand and pressure on the reconfigurable interconnect. On DRAP, the registers used for implicit pipelining are already a hard-wired part of

the datapaths. Explicit pipelining is the only method available on RICA. Because of implicit pipelining, DRAP requires a smaller number of interconnect channels to route pipelined applications than RICA. The subsections that follow will compare the performance of pipelined and non-pipelined datapaths on DRAP and RICA. In DRAP's case, asynchronous cells with latches at the input channels only are used (Figure 7.1).

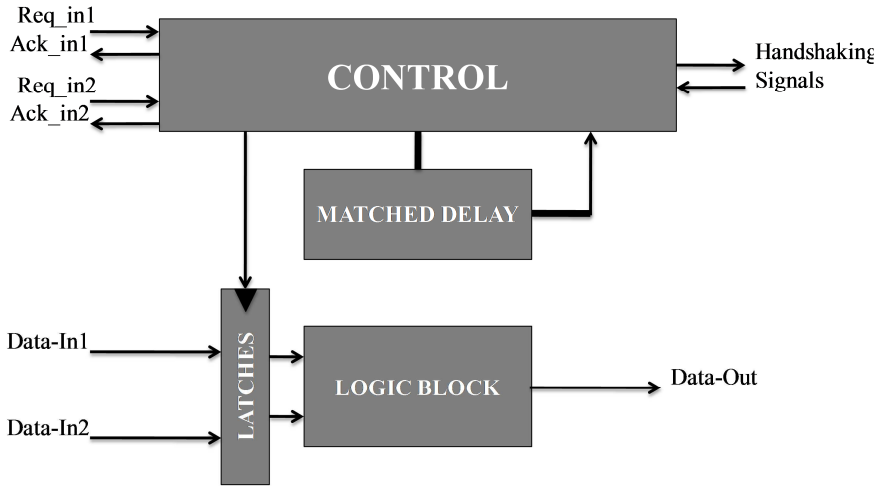


Figure 7.1: Block diagram of an asynchronous cell with latches at the input channels.

7.1.4 Non-pipelined Datapath on RICA vs. DRAP

Figure 7.2 shows two identical datapaths, one on DRAP and one on RICA with no external pipelining added and with cells having the same logic delay of 0.5ns for the SINK and SOURCE cells and 1ns for the ADD cell. The RRC in RICA divides the global clock by a programmed value calculated in software in order to achieve a programmable clock. The critical path in the RICA datapath with no explicit pipelining is the sum of all the cell delays. In this case, the critical path is 5ns. If the RRC divides in 3ns increments, the critical path in practice becomes 6ns. The datapath on DRAP is also not explicitly pipelined. However, the implicit pipelining introduced by the asynchronous cells makes the datapath fully pipelined. The slowest

cell, the ADD cell in this case, dictates the critical path. The iteration rate of the RICA datapath (Figure 7.2) is 1 per 6 ns and the latency is one iteration or 6 ns. Because it is not pipelined, the minimum iteration count for this datapath is one and it does not produce any garbage data.

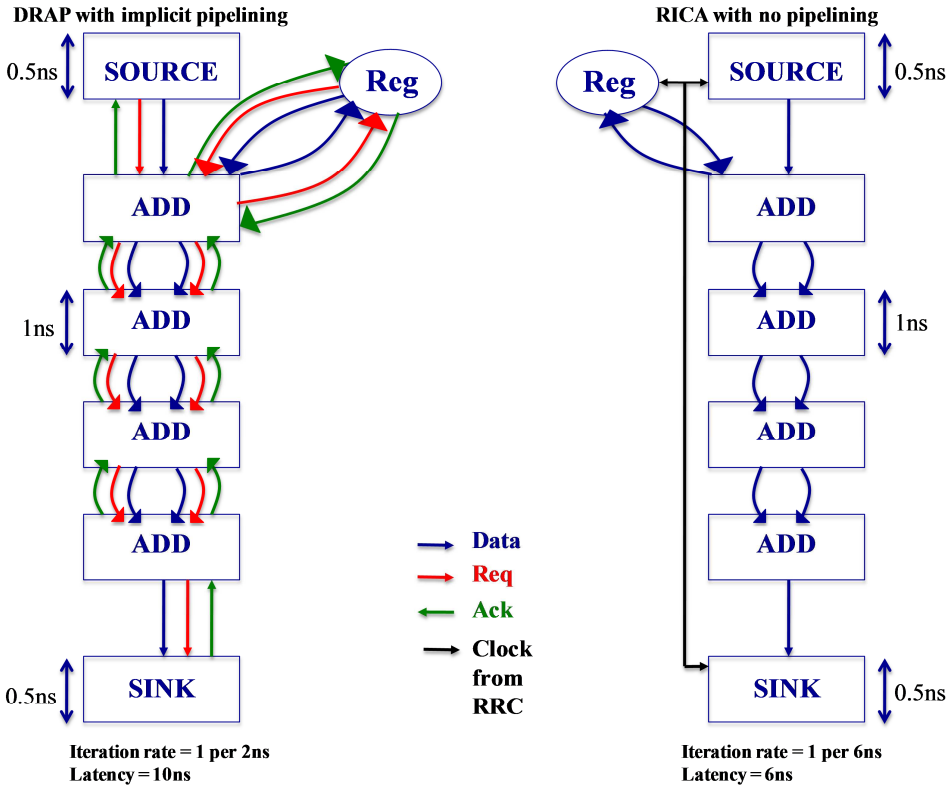


Figure 7.2: DRAP vs. RICA datapath with no explicit pipelining.

On the other hand, the critical path of the DRAP datapath in Figure 7.2 is 1ns plus the delay introduced by the handshaking control. Assuming the handshaking delay is 1 ns, the iteration rate of the DRAP datapath is 1 per 1ns + handshake delay = 1 per 2ns. The latency in this case is five iterations which is equal to 10 ns. The asynchronous cells are designed to wait for inputs, perform computations and then output the data. Because of the nature of the cells, the minimum iteration count of any pipelined DRAP datapath is one and it does not produce any garbage data. As a

result, no special consideration has to be taken for non-kernel steps which will run for only one iteration.

7.1.5 Pipelined Datapath on RICA vs. DRAP

Similarly, Figure 7.3 shows two identical datapaths, one on DRAP and one on RICA both with three-stage explicit pipelining. The critical path in the RICA datapath is now 2ns. Because of the RRC's 3ns division increments, the critical path in practice becomes 3ns. Explicitly pipelining the DRAP datapath in this case does not affect the critical path. It remains 1ns plus the delay introduced by the handshaking control.

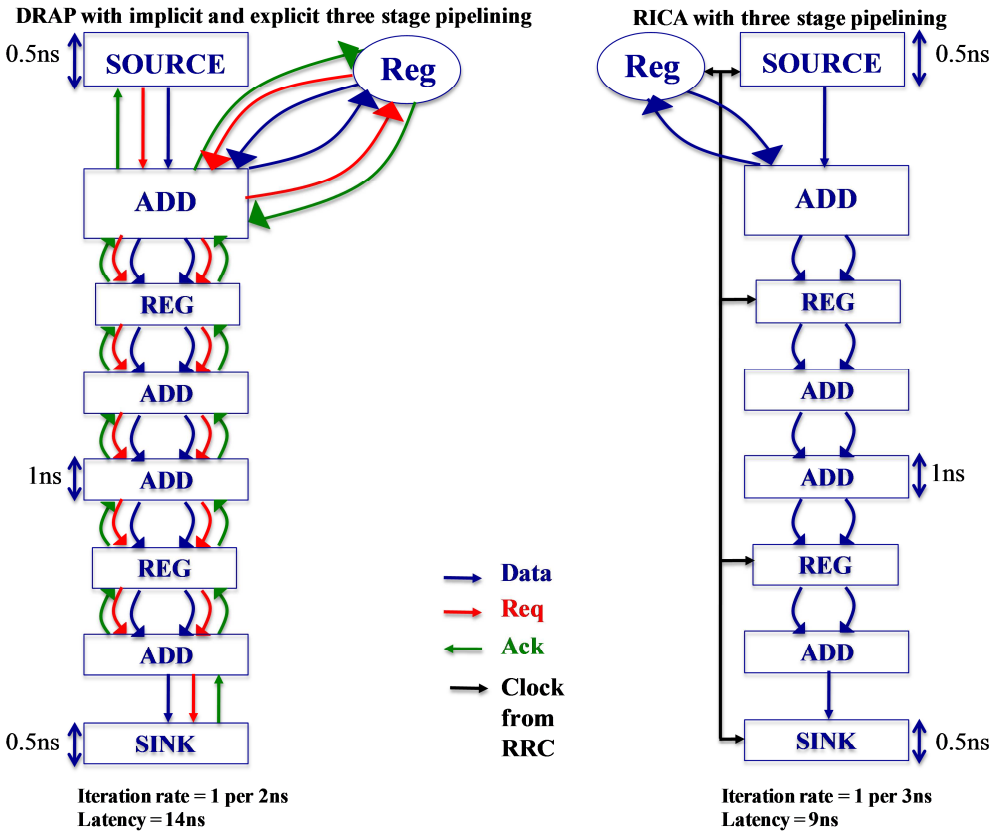


Figure 7.3: DRAP vs. RICA datapath with three-stage explicit pipelining.

The iteration rate for the RICA datapath is now 1 per 3 ns. That of DRAP is unchanged at 1 per 2 ns. The latency of the RICA datapath is three iterations or 9 ns. That of DRAP is now seven iterations or 14 ns. The minimum iteration count for DRAP is still one and it produces no garbage data. However, the RICA datapath now has a minimum iteration count of three where the first two sets of values are garbage.

7.1.6 Limitations of Implicit Pipelining

The property of implicit pipelining in asynchronous cells leads to full pipelining of most datapaths on DRAP. However, there is a class of datapaths where the cells' implicit pipelining does not pipeline it fully. This class is referred to in this thesis as bypass datapaths and defined as a datapath that jumps over several levels in the data flow graph as shown in Figure 7.4a.

In this case, there is a link between the output of the SOURCE cell and the input of the last ADD cell (ADD_4) which bypasses several layers (ADD_1, ADD_2, and ADD_3 cells). Even though each cell contains latches at its inputs, implicit pipelining in this case does not make the datapath fully pipelined. The way this datapath functions is as follows:

The SOURCE initially sends out a request signal; ADD_1 and ADD_4 would both raise their acknowledge signals as a result. The SOURCE will lower its request signal and the two receiving cells will follow suit. This indicates that both ADD_1 inputs and input_1 of ADD_4 have received the first data. The SOURCE now tries to send the second data along with a request signal to indicate its validity. Again the ADD_1 inputs see the request signal and raise their acknowledge signals. ADD_4 cannot raise its acknowledge signal until its second input, input_2 has received its first data, that is until ADD_1, ADD_2, and ADD_3 have finished their computation. As can be seen, handshaking here guarantees correct functioning of the datapath. However, in this case, implicit pipelining alone does not lead to full pipelining. The iteration rate of the datapath in this case is 1 per 3.5ns + 4 x handshake delay = 1 per 7.5 ns. The latency and minimum iteration count are not affected by bypass

datapaths. In this case, the latency is five iterations or 10 ns and the minimum iteration count is one, with no garbage data produced.

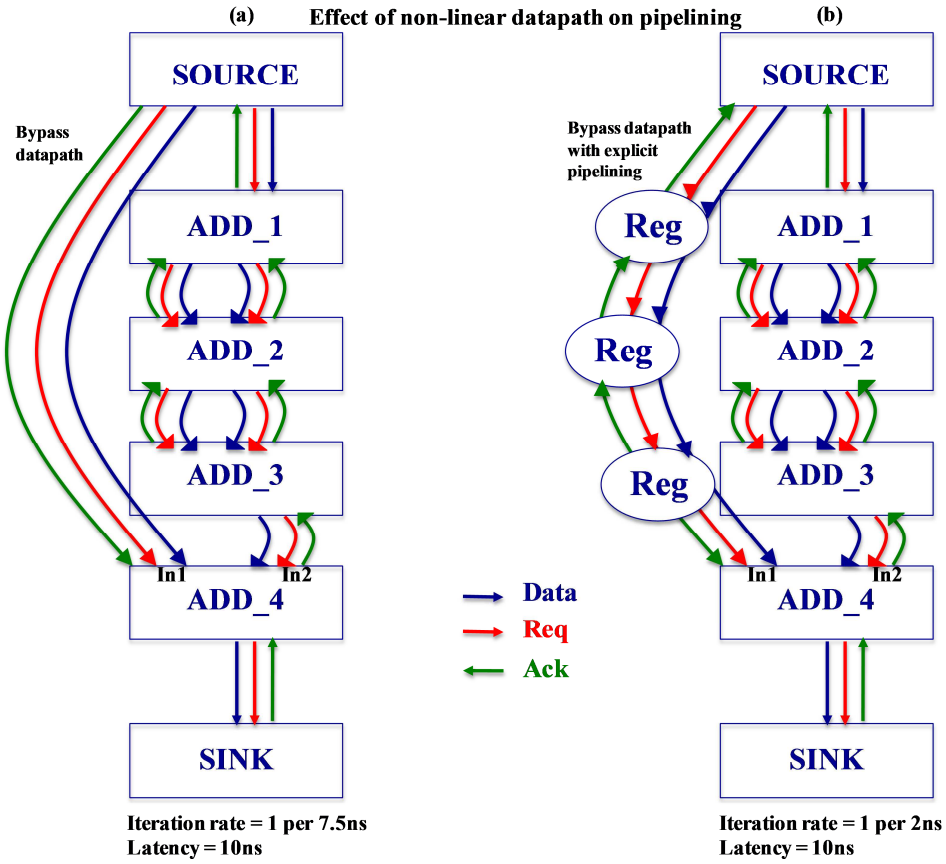


Figure 7.4: (a) Data flow graph of a bypass datapath. (b) Data flow graph of a bypass datapath with explicit pipelining added to the bypass branch to achieve full pipelining.

In order to improve the performance of bypass datapaths, explicit pipelining can be used to reactivate the effect of implicit pipelining. Asynchronous register cells (REG cells) or latches from pipelined interconnects have to be added on the bypass branch of the datapath. The best performance is achieved by adding enough asynchronous latches to match the level of pipelining of the main branch caused by implicit pipelining with that of the bypass branch. In the case of Figure 7.4a, this is achieved by adding three latches to the bypass branch as shown in Figure 7.4b. The result is a

datapath that is fully pipelined. The iteration rate is now 1 per 2 ns while the latency is still five iterations or 10 ns.

If REG cells are added to the bypass branch of the datapath but the number is not matched with the number of latches in the main branch, the datapath will still function correctly and there will be an improvement of performance compared to having no latches at all. This is a useful property of asynchronous logic because in cases where there are not enough resources to balance all branches of a bypass datapath, a throughput improvement is still achieved. Additionally, mismatched pipelining on bypass branches can be used to control the speed of execution of a routed program and hence achieve a trade-off between power and speed.

7.2 Description of the Tool Flow

There are two main components of software support available for DRAP:

Array hardware generation: The tool takes in a Machine Description File (MDF) and generates as output a synthesisable RTL definition of a DRAP core. The RTL definition can be used in a standard SoC tool flow for verification, synthesis, layout, and analysis such as power consumption and timing. The MDF defines the following:

- Cell types and their arity, i.e. the number of operands they take
- Cell instantiations
- Types of operations and how they map to cells
- Interconnect topology

This part of the tool flow was inherited from RICA. The MDF was adjusted to include a new type of cell, the asynchronous REG which can be configured to one of four modes depending on its position in a datapath as described in Section 4.4. The MDF which describes the switchbox was also adjusted to define the REG cell and enable it to be configured at a later stage.

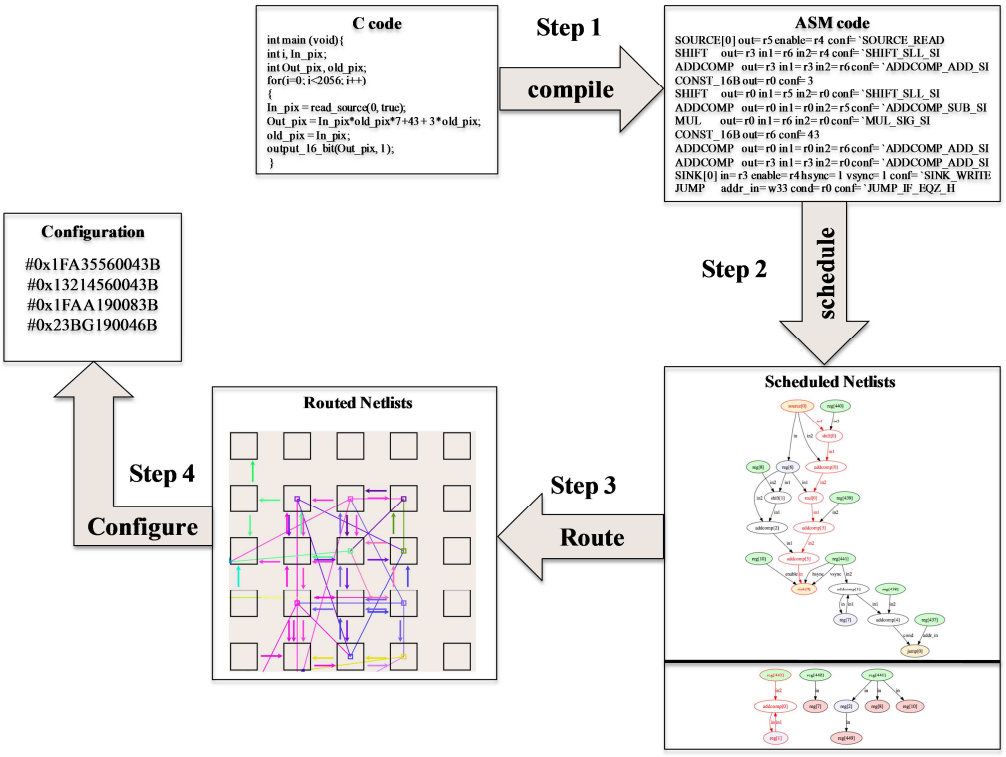


Figure 7.5: Automatic tool flow for DRAP.

The part of the tool that generates the Verilog file of the array was rewritten to include both the additional handshaking signals found in the cells and interconnect and the START signal which is connected to certain cells.

The number of endpoint REG cells is specified and randomly distributed in the array with all relevant connections to the ARC module made. This step was done manually for the generated arrays but will be automated as a next step.

Programming arrays from high-level languages: The granularity and self-reconfigurability of the DRAP architecture make it programmable in a broadly similar way to standard CPUs. This allows existing developments and methodologies such as optimising compilers to be used. Figure 7.5 shows the overall tool flow for programming a DRAP array starting from a high-level C program.

Step 1 - High-Level Compiler: Takes the high-level code and transforms it into an intermediate assembly language. This step is performed by a modified version of the open source GNU Compiler Collection (GCC) [103]. The resulting assembly describes the program as a series of basic blocks, each containing a list of instructions. Each instruction maps directly onto a DRAP asynchronous operational cell. Since GCC has grown up around CPU architectures, its output is based on the supposition that instructions are executed in sequence - i.e. one instruction per cycle; the compiler has no knowledge about the parallelism available on DRAP.

This part of the tool flow was inherited from RICA.

Step 2 - DRAP Scheduling: In this step all the optimisations related to the DRAP architecture are performed. The DRAP scheduler takes the assembly output of GCC and creates a sequence of netlists representing the basic blocks of the program. Each netlist contains a group of instructions that will be executed on DRAP. Temporary registers, allocated by GCC, are replaced by simple wires. The partitioning is done according to dependencies between instructions: dependent instructions are connected in series, whilst independent ones are in parallel. The scheduling algorithm [104] takes into account the operational cell resources and timing constraints in the array to maximise cell occupancy and minimise the longest path delay.

This part of the tool flow was inherited from RICA.

Step 3 - Allocate and Route: For each netlist in the program, the instructions are mapped to physical cells in the array. As there can be numerous available operational cell resources to which a given assembly instruction can be allocated, a tool is provided to minimise the distance between connected cells (similar to a standard place and route tool like VPR [105]).

This part of the tool flow was inherited from RICA with the following adjustments:

- **Routing Modification:** After initial allocate and route, the routed netlists undergo an optimising process in order to get the endpoint cells connected to the ARC (see Section 6.2). For each netlist, the tool measures the distance needed to link the last cell of each datapath with an endpoint cell, while making it the critical path of the netlist. The solution that provides the minimum distance is selected. This is illustrated in Figure 7.6 where one of the steps of a complied program is assumed to have four datapaths. The routing algorithm calculates the best path for the datapaths. At this stage, there is no constraint on the algorithm regarding which datapath should be routed as the critical path. After initial routing, datapath-3 is the critical path. As a next step of this stage, the tool measures which of the four datapaths can be extended to terminate in an endpoint cell while also becoming the critical path. In this example, it turns out to be datapath-1. This part of the tool has not been fully automated. As yet, hand optimisations are still required for each benchmark; however, full automation can be achieved.

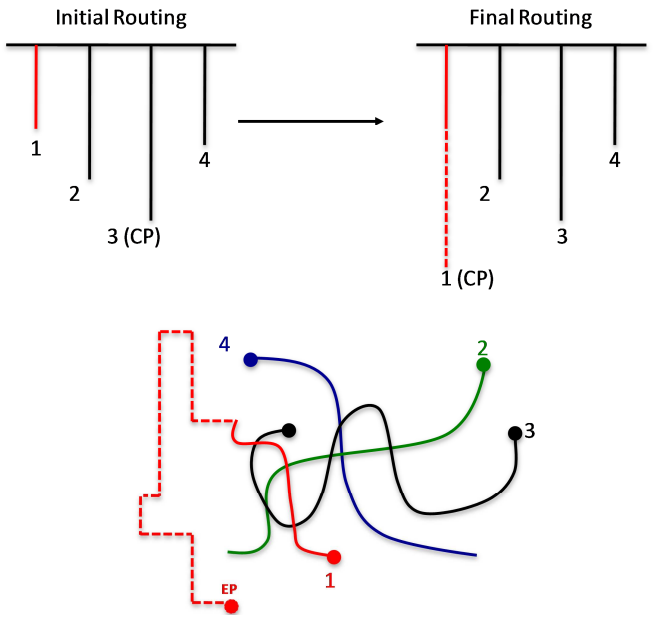


Figure 7.6: Routing modification for DRAP. After initial allocate and route, the routed netlists undergo an optimising process in order to get the endpoint cells connected to the ARC.

- **Pipeline Optimisation:** Because of the bypass datapath problem described in Section 7.1.6, a pipeline optimisation option was added to the tool flow. With this option, the tool identifies all the bypass datapaths in the kernel steps only and optimises them by adding asynchronous latches to the bypass branches such as they match the main branches. The router achieves this by converting delay elements into switchboxes where possible in the case of pipelined interconnect or using REG cells in the case of non-pipelined ones. As yet, this part requires some hand optimisation for any given benchmark; however, full automation can be achieved.

Step 4 - Configuration-Memory: From the mapped netlists, the required content of the configuration memory (program RAM) can be generated.

This part is inherited from the RICA tool flow. It was adjusted to perform the configuration of the REG cells depending on where they are in a datapath.

If the required performance determined by RTL simulation is not met, then the high-level source code can be modified or the mixture of cell resources changed. Adjusting the hardware resources allows the architecture to be tailored to the specific application domain where it is to be used, thus saving area and power. Once array parameters have been decided upon, the generated files can be used for fabrication. If the algorithm continues to change during or after the fabrication process then the code is simply recompiled for those fixed resources.

7.3 Summary

The first part of this chapter introduced the concepts of implicit and explicit pipelining and bypass datapaths and described the roles they play in the performance of asynchronous datapaths. Implicit pipelining allows DRAP to fully pipeline most mapped datapaths. However, there is a class of datapaths where the cells' implicit pipelining does not pipeline it fully. This class was referred to as bypass datapaths and defined as a datapath that jumps over several levels in the data flow graph. For

steps with bypass datapaths, explicit pipelining can be used alongside implicit pipelining to achieve full pipelining.

The next part of the chapter described the different parts of the DRAP tool flow. The designer can customise the number and type of cells and the type of interconnect required. The tool was inherited from the RICA tool flow and adjusted to update cell properties, include new cell types and allow the options of optimising performance in the event of bypass datapaths. The tools were also adjusted to perform the software side of controlling reconfigurability on DRAP.

Chapter 8

Implementations on DRAP

A novel dynamically reconfigurable asynchronous processor called DRAP has been described. It is aimed at high-throughput mobile applications, which increasingly demand a high level of programmability and energy efficiency. DRAP, based on the RICA architecture [4], is also programmed via conventional high-level software languages like C. Additionally, DRAP uses asynchronous logic in the design of its coarse grained heterogeneous cells. Asynchronous logic leads to event-driven energy consumption in DRAP by automatically turning off unused circuits. The elimination of the clock tree also allows DRAP to be more scalable than its synchronous counterparts in terms of power and area overhead.

As shown in Section 7.2, a designer can choose the size of the DRAP array as well as the type, count and position of each of the cells. Additionally, for algorithms that contain bypass datapaths, a designer has a choice between optimised or unoptimised pipelining. Optimised pipelining is an option in the DRAP tool flow which when selected, uses explicit pipelining to balance out bypass datapaths and restore full pipelining of the datapath. Unoptimised pipelining refers to when this option is not

selected; implicit pipelining still exists but does not achieve its full potential of full pipelining in the presence of bypass datapaths (Sections 7.1.6 and 7.2).

Section 1.1 shows that there are two main choices to be made relating to the island-style interconnect structure. They are the number of channels of the switchbox and the types of routing switches (pipelined or non-pipelined). This chapter presents an evaluation of two sample DRAP arrays, each having a different size and interconnect structure.

The sample arrays were tested with algorithms for high throughput streaming applications: the bilinear demosaic algorithm which is representative of the more complex systems found in imaging applications and the FFT algorithm which is a typical system found in mobile applications. The speed and power consumption for each application were calculated and compared to that of RICA and other leading processors.

8.1 Description of Example Algorithms

8.1.1 Bilinear Demosaic Filter

Most digital cameras don't capture Red, Green, and Blue (RGB) values at each pixel and instead use a monochrome sensor overlaid with a mosaicked optical Colour Filter Array (CFA). One of the most used CFAs is the Bayer filter [106]. Each pixel in a Bayer camera measures only one colour channel and captures a raw image that needs to be processed to render it into a viewable format.

Demosaicing (also known as CFA interpolation) is the process of digitally filtering a raw image from the CFA filter to reconstruct an image with R, G, and B values at each pixel. There are many demosaicing algorithms. The simplest one is the nearest-neighbour interpolation [107] which replaces the missing two colour channels at each pixel with those from its nearest neighbours. This method is unsuitable for any

application where quality matters. It provides the correct intensity but produces a 1-pixel shift of hue edges and results in false colours along value edges [107].

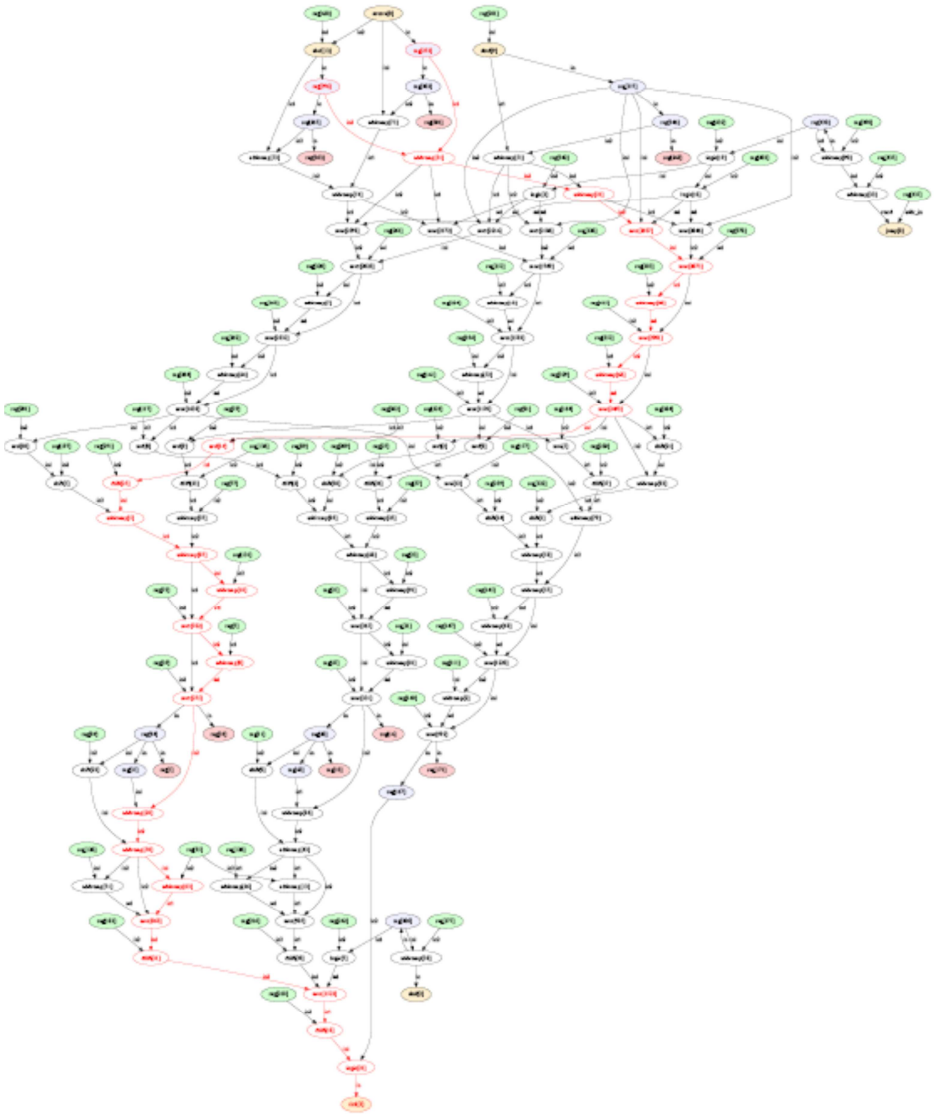


Figure 8.1: The bilinear demosaic filter kernel data flow graph.

Another algorithm is Bilinear Interpolation [94] (also known as Bilinear Demosaic Filter) whereby the missing colour value of a pixel is computed as the average of the two or four adjacent missing colour pixels. Using bilinear interpolation of neighbouring pixels corrects the shift and improves edge colour but blurs high

frequencies. Other more complex interpolation methods such as bicubic interpolation [108], spline interpolation [109] and Lanczos resampling [110] exist.

Table 8.1: Maximum instance counts of the cells for the bilinear demosaic filter.

Cell	Maximum Count
ADDCOMP	38
MUL	8
LOGIC	5
SHIFT	15
MUX	24
REG	87
SOURCE	1
SINK	1
SBUF	2

The test algorithm implemented on DRAP was the 3-line bilinear demosaic filter. It operates on a line of live input along with two lines of history saved in stream buffers. It was chosen as a typical datapath intensive application, so as to be representative of the types of mobile high throughput streaming imaging applications that the architecture was designed for. The 3-line bilinear interpolation is a real-life, high-throughput application normally done on-chip (integrated into the sensor) as part of a custom image signal processing pipeline, used in modern digital cameras and mobile phones. This is a computationally intensive part of a standard Image Signal Processor (ISP). The filter was re-implemented on DRAP, using the C programming language. Software optimisation techniques were used to reduce the filter kernel into a single basic block, small enough to fit onto the target architecture in a single configuration context. The filter kernel data flow graph is shown in Figure

8.1, and the summary of the operations involved is given in Table 8.1. Kernels of imaging filters, such as this, process an entire line of the image each time they are called, so the minimum consecutive iteration count is very high. In the example used, the input is a 16-bit image of size 25 rows x 2056 columns.

8.1.2 FFT

Orthogonal Frequency-Division Multiplexing (OFDM) is a central technology for wireless digital communication applications such as digital television and audio broadcasting, wireless networking and internet access. OFDM is implemented using the Fast Fourier Transform (FFT) algorithms. FFTs are key because they route signals from the baseband to the subcarriers.

As an example, the Digital Video Broadcasting – Satellite services to Handhelds (DVB-SH) transmission system standard [111][112] is designed to deliver video, audio and data services such as mobile television to handheld devices. DVB-SH relies on OFDM to achieve high data rates even in multipath environments. This standard defines four FFT modes for the OFDM receiver: 1K, 2K, 4K and 8K along with several guard intervals. The 8K FFT (8192-point) was chosen for implementation on DRAP because it is a highly computational part of the DVB-SH standard. This standard dictates that the 8K FFT must be performed within 924μs (Appendix B). This FFT is usually implemented on an FPGA or using ASICs, as DSP implementations are too costly in terms of area and power consumption [113].

Equation 8.1: An N-point FFT operation.

$$\boldsymbol{X[k] = \sum_{n=0}^{N-1} x[n].W_N^{nk}, k = 0, 1, \dots, N-1}$$

where the twiddle factor W is:

$$\boldsymbol{W_N^{nk} = e^{-j2\pi nk/N}}$$

Equation 8.1 shows an N-point FFT operation. The main FFT computation is complex and requires a large number of operations. Several algorithms have been designed to reduce its complexity by reducing the number of required computations [114]. For the implementation on DRAP, the chosen algorithm is the Cooley-Tukey decimation-in-time Radix-2 algorithm (Figure 8.2) [102].

$$y_0 = x_0 + x_1 . W_N^r$$

$$y_1 = x_0 - x_1 . W_N^r$$

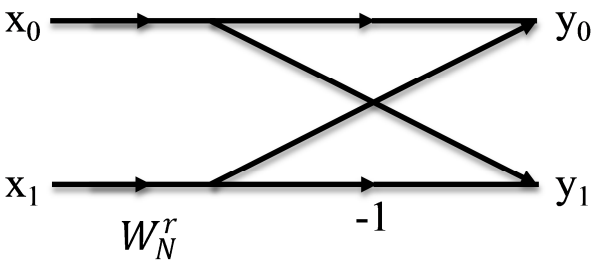


Figure 8.2: Radix-2 complex butterfly computation.

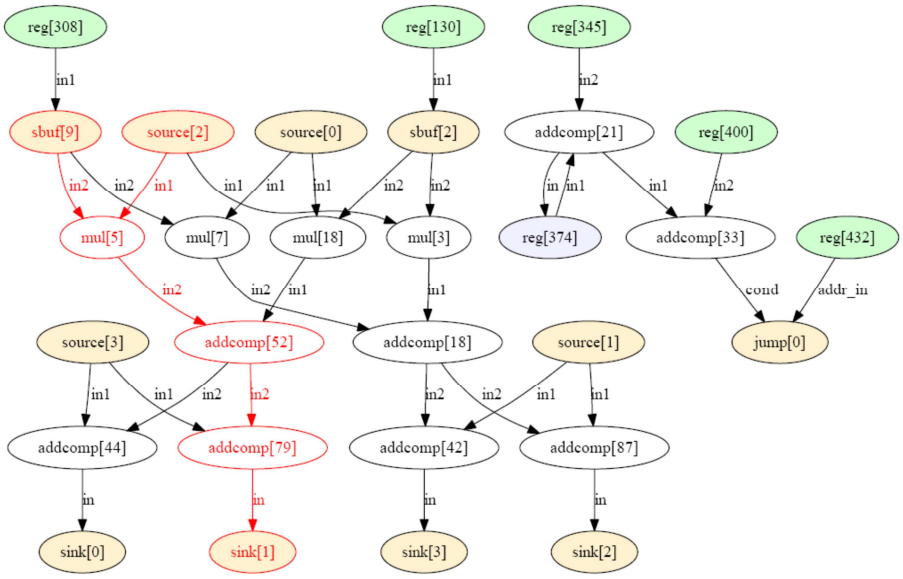


Figure 8.3: The radix-2 FFT butterfly kernel data flow graph.

To compute the 8K FFT (8192-point FFT), 13 stages are required with each stage consisting of 4096 radix-2 butterfly operations. The kernel implementing the radix-2 is hence executed 13x4096 = 53,248 times. A radix-2 butterfly is in effect a 2-point FFT computation; it has two inputs x0 and x1 and two outputs y0 and y1, and uses the twiddle factor WNr (Figure 8.2).

The operations are complex ones, i.e. the numbers have a real and imaginary part. Each butterfly requires one complex multiply and two additions. The real and imaginary parts of each number were treated separately as 18-bit values. Hence 36 bits were used to represent each complex number. The 8K radix-2 FFT requires 53,248 complex multiplications and 106,496 complex additions. Each complex multiplication was implemented using 4 real multipliers and 2 real adders. As a result, the algorithm requires 212,992 real multiplications and 319,488 real additions. Figure 8.3 shows the radix-2 butterfly kernel data flow graph. This kernel runs for 53,248 iterations in order to complete the 8K FFT. The summary of the operations involved in the radix-2 butterfly is given in Table 8.2.

Table 8.2: Maximum instance counts of the cells for the radix-2 FFT butterfly.

Cell	Maximum Count
ADDCOMP	8
MUL	4
LOGIC	1
SHIFT	4
MUX	2
REG	21
SOURCE	4
SINK	4
SBUF	2

8.2 Description of Sample Arrays

Two sample DRAP arrays were designed for testing purposes. The mixture of the operational cells for each array is manually selected to be adequate for general applications. The selection is influenced by the experience of the RICA team of working with imaging and other applications. Other combinations can be chosen to be tailored to an application. The arrays were implemented using a UMC 0.13-μm technology.

8.2.1 20x20, 1-Channel DRAP

The first array contains 400 asynchronous cells as listed in Table 8.3. It uses a single channel interconnect scheme. Four versions of this array were designed: two arrays with pipelined switchboxes, one with 32-bit and the other with 18-bit asynchronous cells (certain mobile imaging applications require 18-bits) and another two arrays with non-pipelined switchboxes, each with a different sized cell (18-bit/32-bit cells). With the selected type of interconnects and operational cells, the reconfigurable core requires a total of 8215 configuration bits. The MUX and REG cells are distributed separately with each cell having its own switchbox.

Table 8.3: Asynchronous operational cells in 20x20 arrays.

Cell	Count	Cell	Count
ADDCOMP	65	LOGIC	20
MUL	20	MUX	65
REG	174	JUMP	1
SHIFT	35	SBUF	8
SINK	4	SOURCE	4

8.2.2 15x15, 5-Channel DRAP

The second sample DRAP array contains 225, 18-bit cells as listed in Table 8.4. It uses 5-channel pipelined switchboxes. With the selected type of interconnects and operational cells, the reconfigurable core requires a total of 16,520 configuration bits. The REG and MUX cells are incorporated into the switchbox (see Section 4.4).

Table 8.4: Asynchronous operational cells in 15x15 arrays.

Cell	Count	Cell	Count
ADDCOMP	97	LOGIC	30
MUL	20	MUX	450
REG	450	JUMP	1
SHIFT	40	SBUF	16
SINK	4	SOURCE	4

8.2.3 Comparing DRAP with Other Architectures

The following hardware architectures were chosen to be compared to the second sample DRAP:

- **RICA_225**: an equivalent 225-cell array based on the RICA architecture in [4] (0.13-µm).
- **ASIC**: an equivalent ASIC design of the tested algorithms (0.13-µm).
- **ARM7-TDMI-S** [115] (0.13-µm).
- **TIC64x 8-way VLIW** [116].

For the evaluation, sample algorithms representative of the more complex systems found in mobile and imaging applications were selected: Bilinear demosaicing [94] and 8K point radix-2 FFT [102]. All the benchmarks are direct unoptimised C representations of the algorithms—all optimisations are left for the C compilers

(Level-3/O3). For each benchmark, the power consumption of each design was calculated for the same throughput.

For the DRAP, RICA_225, and ASIC designs, the power and area were found using post-layout simulations on PrimePower from Synopsys (and post clock tree synthesis for the synchronous designs). The GCC compiler for ARM [103] and the TI compiler [116] were downloaded from their respective websites and used to obtain the number of instructions required for each benchmark to run. With this information, the ARM7 datasheet [115] provides power and area value of the ARM core in 0.13-μm technology, while [118] allows the power consumption of just the datapaths in the TIC64x to be estimated. All these power estimations were obtained at 1.2-V operating voltage and only focus on the energy consumed in the datapath (cells and interconnect) without the memory.

8.3 Results

8.3.1 Pipelined vs. Non-pipelined DRAP

For the first test of the DRAP array, the bilinear demosaic filter was mapped onto both of the 20x20, 32-bit arrays, one with pipelined and the other with non-pipelined 1-channel switchboxes. The aim of this test was to examine the effect of pipelining DRAP's interconnect structure on its area, power and speed. The results are listed in Table 8.5 and Figure 8.4.

With non-pipelined switchboxes, the interconnect delay is variable and depends on the number of switchboxes connecting any two cells. Furthermore, the interconnect delay affects both the rising and falling phases of the request and acknowledge handshaking signals. Since four-phase handshake signalling was used in DRAP, the effect of both the request and the acknowledge interconnect delays on communication between cells is doubled. However, with pipelined switchboxes, the long interconnect delays are broken down and fixed at a delay of one switchbox. This

reduces the effect of interconnect delay doubling which nonpipelined interconnects suffer from (see Section 5.2). With pipelined interconnects, the critical path of any datapath mapped on the array is the delay of the slowest cell in the longest datapath (including handshaking delays) plus the delay of one switchbox.

For the algorithm mapped on the non-pipelined interconnect array with unoptimised pipelining (see Section 7.2), the connection between two cells of the kernel's longest datapath passes through 30 switchboxes. This long connection coupled with the interconnect delay doubling effect leads to the large critical path and hence low throughput. With optimised pipelining, the long connection is reduced to 20 switchboxes. However, the interconnect delay doubling effect limits the effect this reduction has on the throughput.

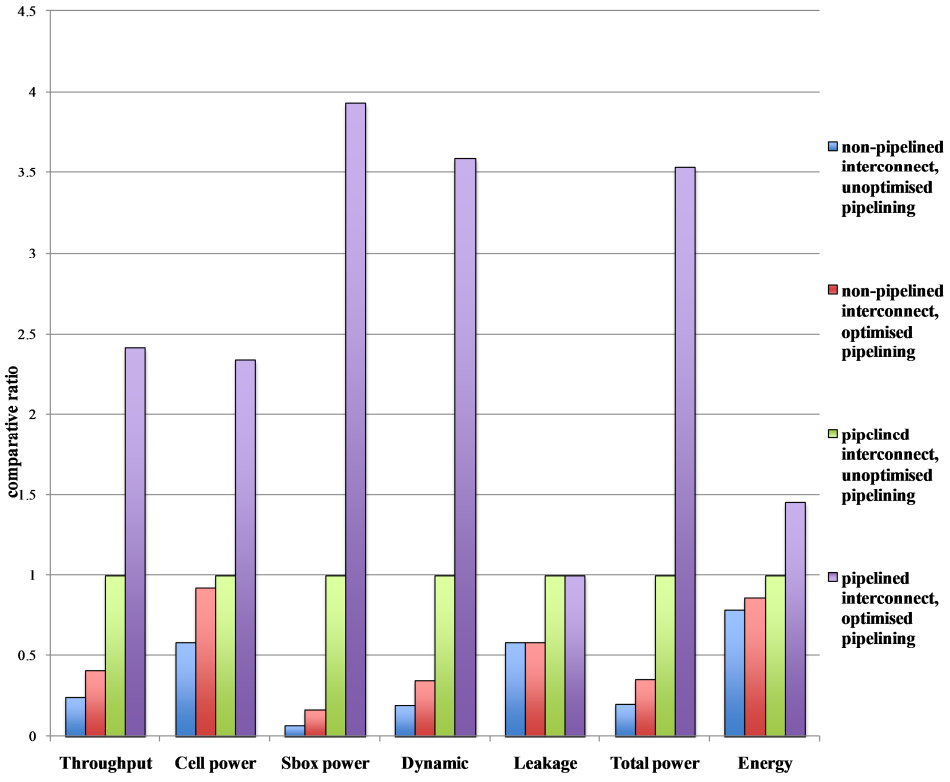


Figure 8.4: Normalised throughput, power and energy consumption graph of the bilinear demosaic benchmark on a pipelined and non-pipelined 20x20, 1-channel DRAP.

For the algorithm mapped on the pipelined interconnect array, the interconnect delay is now fixed at one switchbox. In theory, an algorithm mapped on this array should have a critical path comprising of the delay of the slowest cell plus that of one switchbox. However, the mapped bilinear demosaic algorithm contains bypass datapaths.

With unoptimised pipelining, these datapaths reduce the effect of the implicit pipelining and hence the throughput is slower than the theoretical maximum. With optimised pipelining, explicit pipelining is added and the delay is reduced to its minimum. As a result, the algorithm runs at its maximum possible throughput on this array.

Table 8.5 shows that, as expected, pipelined switchboxes enable DRAP to achieve higher throughputs than non-pipelined ones. The maximum throughputs the array with non-pipelined interconnect achieved are 8.5 and 14.1 Mpixels/s with unoptimised and optimised pipelining respectively. On the other hand, the maximum throughputs the array with pipelined interconnect achieved are 34.5 and 83.3 Mpixels/s with unoptimised and optimised pipelining respectively. This is around 4x and 6x increase in throughput for unoptimised and optimised routing respectively.

However, with pipelined switchboxes, there is an increase in switching activity associated with the additional asynchronous latches in the interconnect. This resulted in an increase of around 1.7x and 2.5x in cell power and around 14.5x and 23.8x in switchbox power with unoptimised and optimised pipelining respectively. This translated to an increase in total power consumption of around 5x and 10x with unoptimised and optimised pipelining respectively.

Nonetheless, the increase in power was offset by the reduction in processing time. As a result, the array with pipelined interconnects consumed only around 1.3x and 1.7x more energy than the array with non-pipelined interconnects, with unoptimised and optimised pipelining respectively.

The ratio of the switchbox power to cell power is 0.6 and 5 for non-pipelined and pipelined array respectively (both with optimised pipelining). For the non-pipelined array, the cost of reconfigurability is low and the cells consume 65% of the total power. When moving to a pipelined array, the cost of reconfigurability increases significantly where 84% of the total power is now consumed by the switchboxes.

By moving from a non-pipelined array to a pipelined one, there is an increase in both the dynamic and leakage power of the array. The ratio of increase of leakage power as a result of moving from a non-pipelined to a pipelined array is 1.7 (with optimised pipelining).

On the other hand, the ratio of increase of dynamic power is 10.3 (with optimised pipelining). The change in leakage power is 6x less than the change in dynamic power. As a result, the DRAP pipelined array could be suitable for modern processes (22nm and below) where leakage dominates total power consumption. However, a more detailed study is needed to test the effect of variability in deep sub-micron processes on asynchronous circuits using bundled data encoding. This is beyond the scope of this thesis.

Table 8.5: Bilinear demosaicing (1 line x 2056 pixels) on 20x20, 1-channel 32 bit array, pipelined vs. non-pipelined interconnect.

Algorithm	Interconnect	Schedule and Route	Critical path (ns)	Throughput (Mpixels/s)	Cell power (mW)	Sbox power (mW)	Dynamic (mW)	Leakage (mW)	Total power (mW)	Energy (μJ)
Bilinear Demosaic	non-pipelined	unoptimised	117	8.5	7.97	2.85	10.13	0.69	10.82	2.60
		optimised	71	14.1	12.67	6.85	18.83	0.69	19 52	2.85
	Pipelined	unoptimised	29	34.5	13.78	41.49	54.08	1.19	55.27	3.30
		optimised	12	83.3	32.19	163.11	194.11	1.19	195.30	4.80

Area Analysis

The area analysis of the non-pipelined and pipelined arrays is summarised in Figure 8.5. All the area figures are for post-routing. For the non-pipelined DRAP array, the total area was 5.45mm^2. The cell area was 2.25mm^2 whereas the switchbox area was 3.20mm^2. The cell area forms around 41% of the total area in this case and the switchboxes form the remaining 59%. With the pipelined array, the cell area was unchanged but the switchbox area increased to 4.79mm^2. The switchboxes now formed 68% of the total area of 7.04mm^2.

The pipelined array comes at the expense of a 33% area increase over the non-pipelined array. However, the benefits associated with the pipelined interconnect outweigh the area penalty. By moving from the non-pipelined to the pipelined array, the throughput increase was around 6x whereas the area increase was 1.3x. The ratio of the throughput increase divided by the area increase is 4.6. Hence, there is an overall gain of using a pipelined array.

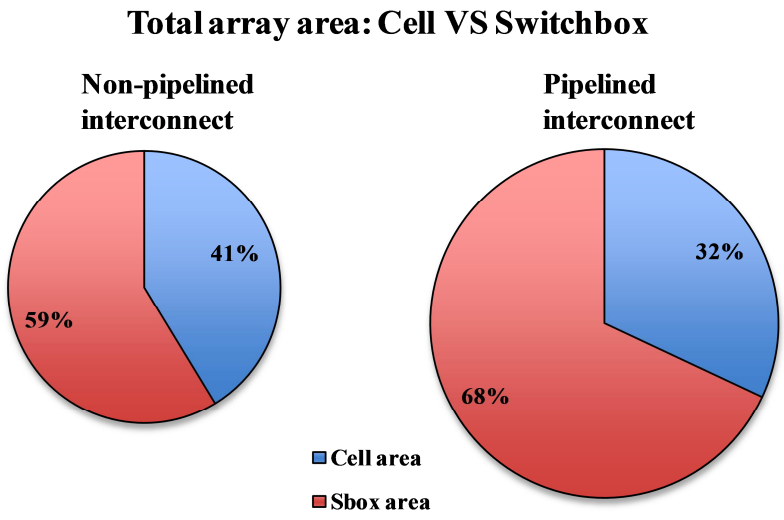


Figure 8.5: Breakdown of the total area of the 20x20, 1-channel array with both non-pipelined and pipelined interconnect as percentage of cell and switchbox (Sbox) area. The total area of the array with pipelined interconnect is 33% larger than the array with non-pipelined interconnect.

8.3.2 Single-channel vs. Multi-channel DRAP

As discussed in Section 5.2.1, an important consideration about the structure of the interconnect design is how likely it will be that a datapath is routable at the end of the tool flow. The routability of the interconnect networks greatly affects the gate utilisation and the speed of DRAP. The initial DRAP design uses a single channel switchbox, i.e. one input and output channel on each of its four sides and allows a maximum of 12 different path routes within it. To improve routability, the number of channels on the interconnect design were increased. A 5-channel pipelined switchbox with five input and output channels on each of its four sides was designed.

For the second test of the DRAP array, the bilinear demosaic filter was mapped onto both the 20x20, 18-bit array with pipelined 1-channel switchboxes and the 15x15, 18-bit array with pipelined 5-channel switchboxes. The aim of this test was to examine the effect different interconnect channel width have on routability and speed. The results are listed in Table 8.6 and Figure 8.6.

With the 5-channel array, the routing step of the DRAP tool flow was around 8x faster than with the 1-channel array. Additionally, when the optimised pipelining option was selected, the algorithm failed to route on the 1-channel array. The netlist had to be adjusted in order for it to route with optimised pipelining. The number of registers needed to perform the optimisation through the use of explicit pipelining was manually reduced. No such problems occurred with the 5-channel DRAP array.

Table 8.6: Bilinear demosaicing (1 line x 2056 pixels) on 20x20, 1-channel 18 bit pipelined array, pipelined vs. on 15x15, 5-channel 18 bit pipelined array.

Algorithm	Array	Schedule and Route	Critical path (ns)	Throughput (Mpixels/s)	Cell power (mW)	Sbox power (mW)	Dynamic (mW)	Leakage (mW)	Total power (mW)	Energy (μJ)
Bilinear Demosaic	5 channel 15x15 pipelined	unoptimised	11.2	89.2	13.3	62.9	74.5	1.7	76.2	1.75
		optimised	8	125	17.8	86.8	102.9	1.7	104.6	1.72
	1 channel 20x20 pipelined	unoptimised	33	30.3	8.5	26.0	33.7	0.8	34.5	2.34
		optimised	8	125	33.4	102.1	134.7	0.8	135.5	2.23

Table 8.7: Bilinear demosaicing (1 line x 2056 pixels) on 15x15, 5-channel 18 bit array pipelined: dynamic and leakage power consumption.

Algorithm	Schedule and Route	Critical path (ns)	Throughput (Mpixels/s)	Switching (mW)	Internal (mW)	Dynamic (mW)	Leakage (mW)	Total power (mW)
Bilinear Demosaic	Unoptimised	11.2	89.2	24.9	49.6	74.5	1.7	76.2
	Optimised	8	125	34.4	68.5	102.9	1.7	104.6

An interesting observation from Table 8.6 is that the total power consumption of the 1-channel array was around 24% more than that of the 5-channel array, when both were running at the same throughput (with optimised pipelining). The main reason for this is that with the 1-channel array, it was harder to route the algorithm and more switchboxes had to be active during the kernel step than in the case of the 5-channel array. The 1-channel array required 365 active switchboxes whereas the 5-channel array required 198 active switchboxes during the kernel step (with unoptimised pipelining).

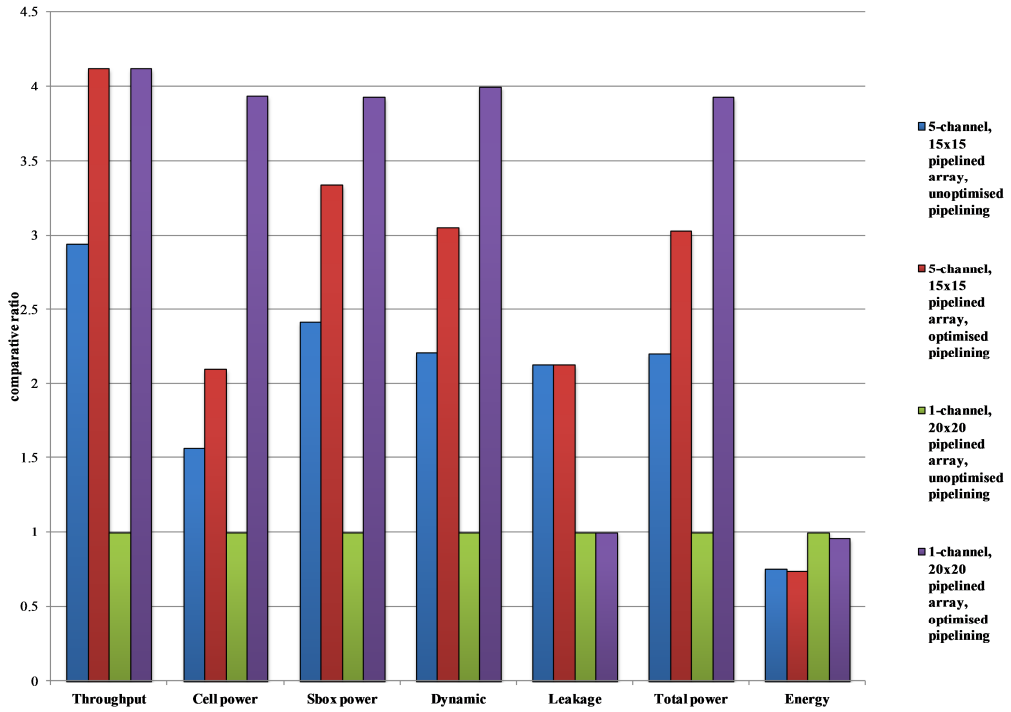


Figure 8.6: Normalised throughput, power and energy consumption graph of the bilinear demosaic benchmark on a 15x5, 5-channel pipelined DRAP array and a 20x20, 1-channel pipelined one.

8.3.3 Bilinear Demosaic on 15x15 DRAP

Focusing on the bilinear demosaic's implementation on the 15x15, 5-channel array with pipelined interconnect as presented in the first row of results in Table 8.6. The

scheduled netlist contained bypass datapaths. The critical path of the kernel step was 11.2ns giving an average throughput of 89.2Mpixels/s. After performing scheduling and routing optimisation, the critical path was reduced to 8ns, which is equivalent to the delay of the slowest cell in the longest path (ADDCOMP in this case) plus the delay of one switchbox. Hence, the algorithm achieved a throughput of 125Mpixels/s.

Table 8.7 and Figure 8.7 show a breakdown of the power consumption of the sample bilinear demosaic algorithm running with optimised pipelining. Dynamic power constitutes around 97.8% of the total power where only around 2.2% is leakage power. Dynamic power is divided into switching and internal power (Figure 8.8), which form 33.4% and 66.6% of the total power respectively.

Figure 8.9 shows the distribution of power between the asynchronous operational cells and the asynchronous 5-channel switchboxes. Most of the power is dissipated by the interconnects. The cell and switchbox power form 17% and 83% of the total power consumption with optimised pipelining and 17.5% and 82.5% with unoptimised pipelining respectively.

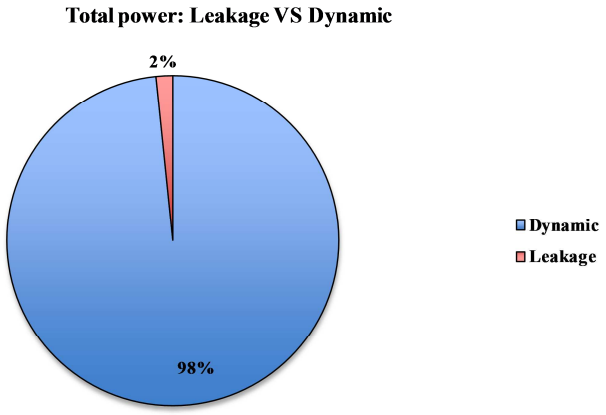


Figure 8.7: Breakdown of the total power consumption of the bilinear demosaic benchmark on the 15x15, 5-channel pipelined DRAP array, as a percentage of leakage and dynamic power.

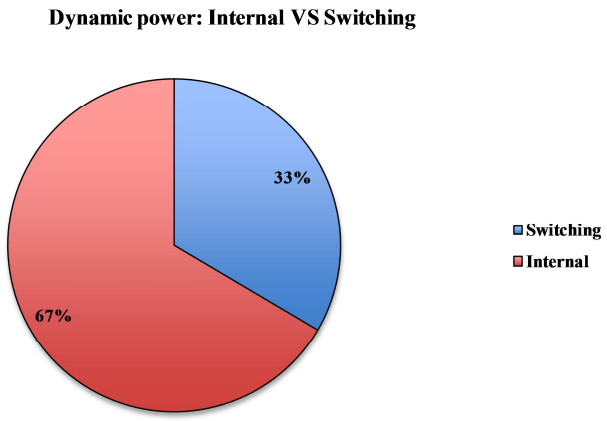


Figure 8.8: Breakdown of the dynamic power consumption of the bilinear demosaic benchmark on the 15x15, 5-channel pipelined DRAP array, as a percentage of internal and switching power.

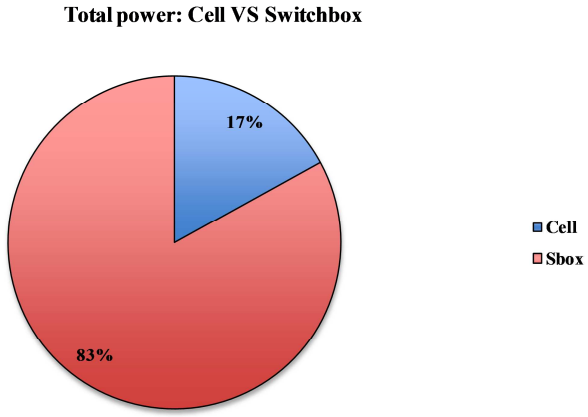


Figure 8.9: Breakdown of the total power consumption of the bilinear demosaic benchmark on the 15x15, 5-channel pipelined DRAP array, as a percentage of cell and switchbox (Sbox) power.

8.3.4 FFT on 15x15 DRAP

So far, the DRAP array was evaluated using the bilinear demosaic algorithm, which is representative of the more complex systems found in imaging applications. In this subsection, the results from mapping an 8K radix-2 based FFT algorithm on the 15x15, 5-channel pipelined array are shown. This algorithm is chosen as a sample of a typical system found in mobile applications. The results are listed in Figure 8.10 through Figure 8.12 and Table 8.8 through Table 8.9.

The radix-2 FFT algorithm was compiled and mapped onto the array. The scheduled netlist contained no bypass datapaths so no scheduling and routing optimisation was necessary. Because the array used pipelined interconnects, the critical path of the algorithm was expected to be that of the slowest cell in the longest path (ADDCOMP in this case) plus the delay of one switchbox. The test validated the expectation as the critical path was measured to be 8 ns, giving a throughput of 250 Msamples/s.

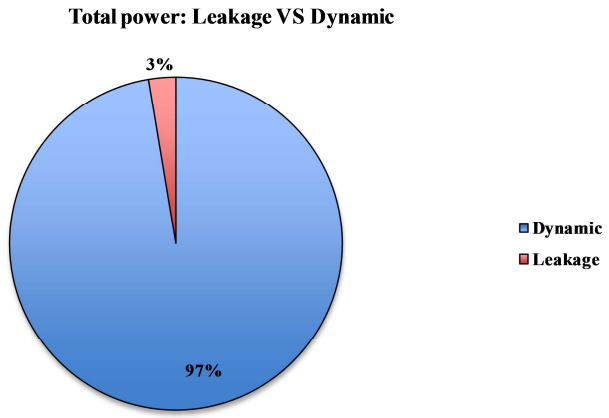


Figure 8.10: Breakdown of the total power consumption of the 8K radix-2 FFT benchmark on the 15x15, 5-channel pipelined DRAP array, as a percentage of leakage and dynamic power.

Table 8.8 and Figure 8.10 show a breakdown of the power consumption of the sample FFT algorithm. Dynamic power constitutes around 97.4% of the total power where only around 2.6% is leakage power. Dynamic power is divided into switching

and internal power (Figure 8.11), which form 35.3% and 62.1% of the total power respectively.

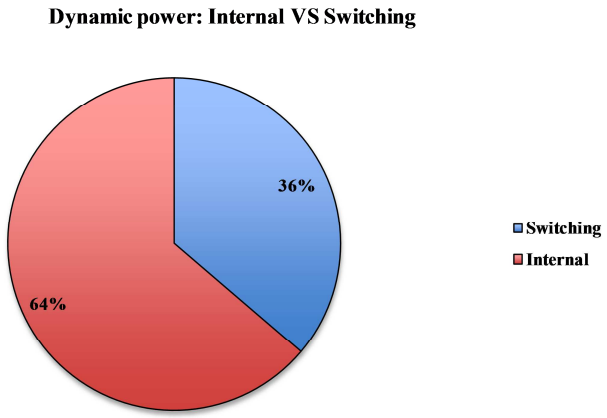


Figure 8.11: Breakdown of the dynamic power consumption of the 8K radix-2 FFT benchmark on the 15x15, 5-channel pipelined DRAP array, as a percentage of internal and switching power.

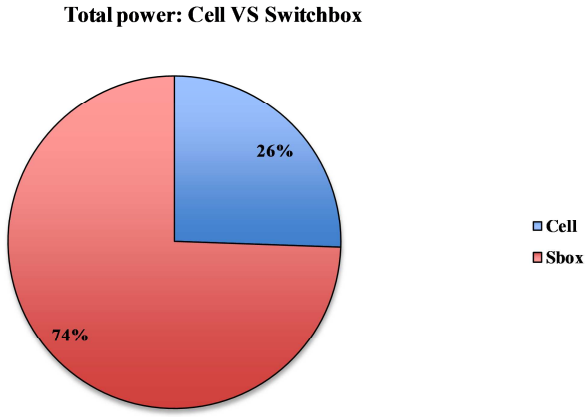


Figure 8.12: Breakdown of the total power consumption of the 8K radix-2 FFT benchmark on the 15x15, 5-channel pipelined DRAP array, as a percentage of cell and switchbox (Sbox) power.

Table 8.9 and Figure 8.12 show the distribution of power between the asynchronous operational cells and the asynchronous 5-channel switchboxes. Most of the power is dissipated by the interconnects. The cell and switchbox power form 25.6% and 74.4% of the total power consumption. Again, the dynamic power dominates the total power. Therefore DRAP is likely to scale well to smaller technology process nodes, where leakage power becomes increasingly dominant.

Table 8.8: 8K radix-2 FFT on 15x15, 5-channel 18 bit array pipelined: dynamic and leakage power consumption.

Algorithm	Critical path (ns)	Throughput (Msamples/s)	Switching (mW)	Internal (mW)	Dynamic (mW)	Leakage (mW)	Total power (mW)
FFT	8	250	8.0	14.1	22.1	0.6	22.7

Table 8.9: 8K radix-2 FFT on 15x15, 5-channel 18 bit array pipelined: cell and switchbox (Sbox) power consumption.

Algorithm	Critical path (ns)	Throughput (Msamples/s)	Cell power (mW)	Sbox power (mW)	Total power (mW)
FFT	8	250	5.8	16.9	22.7

8.3.5 DRAP vs. Other Architectures

In this subsection the DRAP array is compared to different architectures. The sample algorithms of 8K radix-2 FFT and bilinear demosaic are mapped onto the 15x15, 5-channel DRAP array and onto the equivalent RICA_225 array. An ASIC of each of the sample algorithms was designed and tested in simulation. The power and speed values for the algorithms running on the ARM7 and the TIC64x were estimated from relevant datasheets (Section 8.2.3).

From Table 8.10 through Table 8.11 as well as Figure 8.13 through Figure 8.14, it can be seen that DRAP achieves better performance for all benchmarks used than the conventional ARM7 CPU and the TIC64x VLIW. The sample DRAP consumes between 17 and 20 times less power than the ARM7 for the algorithms running at the same throughput. It also consumes between 25 and 29 times less power than the TIC64x for the algorithms running at the same throughput. However, it should be pointed out that the proposed DRAP is capable of achieving a much higher throughput performance than the ARM7 for all benchmarks and the TIC64x for the bilinear demosaic benchmark. The maximum frequency of the ARM7 chip is 110MHz and that of TIC64x is 600MHz. In measuring the power for the same throughput as DRAP, it is assumed that several ARM7 chips are working in parallel in the case of the both sample algorithms and several TIC64x chips working in parallel in the case of the bilinear demosaic algorithm.

A big part of the power reductions achieved over the DSP systems are savings gained by eliminating the register files and having distributed registers. Compared to an equivalent ASIC design of each algorithm, DRAP consumes only between 2.3 and 3.4 times more power.

To evaluate the benefit of using an asynchronous substrate, DRAP was compared to the equivalent RICA_225 array. The DRAP design achieved a power consumption reduction of up to 47% (or around 1.9x less power) for the bilinear demosaicing

algorithm running at the same throughput. It achieved a power consumption reduction of up to 36.5% (or around 1.6x less power) for the 8K radix-2 FFT algorithm running at the same throughput. This is a direct result of the lower level of switching power in the asynchronous design due to its inherent fine-grain “clock gating”. Also the ease of pipelining associated with the DRAP architecture allows it to pipeline fully most applications and achieve higher throughputs than RICA. For the example of the bilinear demosaic, the minimum critical path for the mapped algorithm on RICA_225 was 22ns giving a maximum throughput of 45Mpixels/s. This is around 2.8x lower than the maximum throughout achieved by DRAP for the same algorithm.

For the example of the 8K radix-2 FFT, the minimum critical path for the mapped algorithm on RICA_225 was 10ns giving a maximum throughput of 200Mpixels/s. This is 20% lower than the maximum throughout achieved by DRAP for the same algorithm.

The post-routing area analysis of DRAP and RICA_225 is shown in Table 8.12. The reduction in power of DRAP over RICA_225 comes at a cost of around 22% increase in total array area. However, as shown in Section 3.2.4, a 5-channel DRAP requires a smaller number of configuration bits than an equivalent RICA based array (11% less configuration bits). Additionally, the DRAP architecture requires at least one less interconnect channel to route the same algorithm fully pipelined (Section 3.2.1). Comparing a 4-channel DRAP array with an equivalent RICA based 5-channel array, the DRAP array is around 4.4% larger. However, it requires 26% less configuration bits (Table 8.12). It would also consume even less power than shown in Table 8.10 and Table 8.11 due to a reduction in leakage power associated with a reduced gate count.

Table 8.10: Bilinear demosaicing on 15x5, 5-channel 18-bit array with pipelined interconnect running 1 line of 2056 pixels.

Algorithm	Description	Critical path (ns)	Throughput (Mpixels/s)	Required Frequency (GHz)	Power (mW)
DRAP	unoptimised	11.2	89.2	NA	76.2
	optimised	8	125	NA	104.6
RICA_225	matched speed	8	125	NA	196.2
	maximum speed	22	45	NA	74.8
ASIC	matched speed	8	125	NA	45.6
ARM7	matched speed	NA	125	18.9	2075.5
TIC64x	matched speed	NA	125	2.5	2993.73

Table 8.11: 8K radix-2 FFT on 15x5, 5-channel 18-bit array with pipelined interconnect.

Algorithm	Description	Critical path (ns)	Throughput (Msamples/s)	Required Frequency (GHz)	Power (mW)
DRAP	maximum speed	8	250	NA	22.7
RICA_225	matched speed	8	250	NA	35.8
	maximum speed	10	200	NA	29.3
ASIC	matched speed	8	250	NA	6.8
ARM7	matched speed	NA	250	3.6	392.8
TIC64x	matched speed	NA	250	0.47	558.7

Table 8.12: DRAP and RICA_225 array area and number of configuration bits.

Array	Description	Cell Area (mm^2)	Sbox Area (mm^2)	Total Area (mm^2)	Configuration Bits
RICA_225	5-channel	1.10	6.48	7.58	18660
DRAP	5-channel	1.35	8.38	9.73	16520
DRAP	4-channel	1.35	6.58	7.93	13820

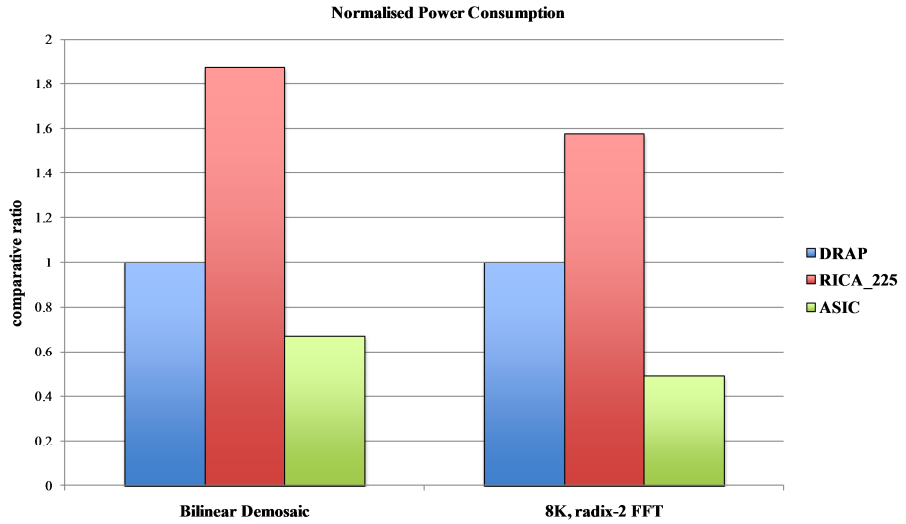


Figure 8.13: Normalised power consumption (with optimised pipelining and matching throughput) of the benchmarks on DRAP, RICA_225, and ASIC.

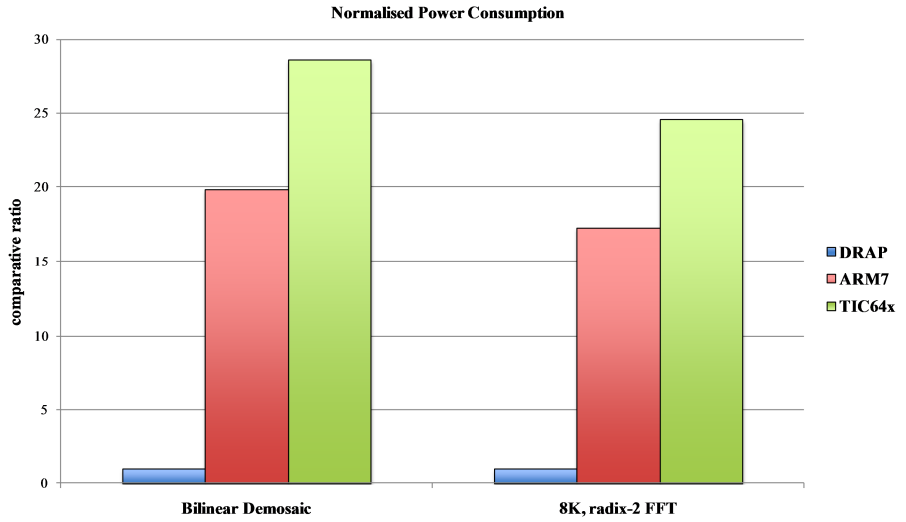


Figure 8.14: Normalised power consumption (with optimised pipelining and matching throughput) of the benchmarks on DRAP, ARM7, and TIC64x.

8.4 Summary

This chapter provided an evaluation of the DRAP architecture and a comparison against RICA, ASICs and other leading programmable architectures. It began by describing the benchmarks that were used to test DRAP. The sample algorithms were chosen because they are representative of typical systems found in mobile and imaging applications.

Several versions of the DRAP array were designed and compared with each other. The results are summarised in Table 8.13:

- DRAP with pipelined interconnects were found to achieve up to 6x higher throughput than one with non-pipelined interconnects. This came at a cost of up to 10x increase in total power consumption or 1.7x increase in energy consumption.
- DRAP with a single channel interconnect was compared to one with a 5-channel interconnect. The multi-channelled DRAP was found to have a higher routability than the single channel DRAP.

The proposed DRAP architecture was also compared to the RICA architecture, ASIC implementations and DSP and VLIW technologies. The results of the comparison are summarised in Table 8.14. When running the same benchmarks at the same throughput, DRAP consumed up to 1.9x less power than an equivalent RICA based design and up to 3.4x more power than an ASIC implementation of the benchmark. It consumed up to 20x and 29x less power than an ARM7 DSP and a TIC64x VLIW respectively.

Additionally, the proposed DRAP architecture was capable of achieving a much higher throughput performance than the ARM7 for all benchmarks and the TIC64x for some. Compared to an equivalent RICA, DRAP achieved throughputs 2.8x larger. The DRAP array however was up to 22% larger in area.

Table 8.13: The effect of using pipelined and multi-channel interconnects on DRAP.

DRAP: pipelined vs. non-pipelined interconnect
• Higher throughput • Higher energy and power consumption • Larger area
DRAP: Multi-channel v.s. single-channel interconnect
• Higher routability • Potentially lower power • Larger area

Table 8.14: Comparing DRAP to RICA and other leading technologies.

DRAP vs. RICA
• Lower power: up to 1.9x • Increased scalability and routability • Less configuration bits (i.e. smaller area for program RAM): up to 26% • Larger area: up to 22% • Easier to pipeline hence higher throughputs more easily achieved: up to 2.8x

DRAP vs. DSP/VLIW
• Lower power: up to 20x/29x • Distributed registers as opposed to centralised register file • Distributed data memory access • Higher throughput • Larger program size

DRAP vs. ASIC
• Flexible • Programmable using high-level C language • Higher power: up to 3.4x • Larger area: up to 17x • ASICs should be able to achieve a higher degree of parallelism due to reduced area limits • If DRAP replaces several hardwired Intellectual Properties (IPs), its distributed memory removes the need for a shared bus and hence reduces power

Chapter 9

Conclusions

The overall objective of the work presented in this thesis was to develop and evaluate a reconfigurable processor as a solution to the increasing demands of mobile applications for high-throughput, programmability, and energy efficiency. Upon studying different programmable architectures, ranging from microprocessors to fine-grained and coarse-grained reconfigurable computers (Section 2.3), CGRAs were found to be the most promising for the growing realm of multimedia and streaming applications. In particular, the family of RICA architectures stand out among other coarse-grained reconfigurable computers in their ability to provide a high level of programmability, energy efficiency and performance suitable for the demands of compute-intensive mobile applications such as multimedia (Sections 2.3 and 2.5). The RICA family of processors are characterised by the following properties:

- The processor consists of an array of heterogeneous coarse-grained combinatorial and synchronous cells that can be dynamically reconfigured on every cycle.

- The processor can be programmed via conventional high-level software languages like C.
- The cells in the array support operations similar to those in a typical RISC instruction set.
- The processor uses the concept of distributed registers where a significant fraction of the array cells are clocked registers.
- The reconfigurable array is in control of its own reconfiguration.

However, because RICA uses the concept of distributed registers to pipeline kernels, it requires a large clock tree to synchronise them. Additionally, as the number of cells increase, Rent's rule leads to a disproportionate increase in the interconnect and number of registers, in order to maintain the ability to route and to pipeline. The increase in registers requires a similar increase in the clock tree, which quickly dominates the power consumption. This gives RICA poor scalability in terms of area and power.

The coarse-grained nature of the RICA architecture along with its large clock tree made it a great candidate for applying asynchronous design techniques, in order to improve energy efficiency and scalability.

This thesis presented and evaluated a novel Dynamically Reconfigurable Asynchronous Processor called DRAP, aimed at high-throughput mobile applications. DRAP was based on the RICA family of processors in order to achieve a high level of programmability. It was designed using asynchronous design techniques in order to achieve a lower power consumption and higher throughput than RICA and other leading processors.

The sections of this chapter correspond to each of the main distinguishing features of DRAP as listed in Chapter 1: event-driven energy consumption, implicit pipelining, scalability and reduced program size. They show how the theories and results presented in the thesis validate each of the features. The chapter concludes with an

overall assessment of what the work achieves and presents ideas for future development of the work.

9.1 DRAP Features

9.1.1 Scalability

Most mobile applications require large kernel steps. In order to run these applications efficiently and achieve high throughputs, the target reconfigurable processor must be large enough to map the entire kernel onto one configuration context. This makes it desirable and highly beneficial to increase the number of operational cells and hence size of a reconfigurable processor aimed at mobile applications.

For a synchronous reconfigurable processor like RICA, as its core gets bigger, the number of sequential elements (primarily registers) in the array must be significantly increased in order to maintain routability and pipelining. As a result, the area of the clock tree would increase and it would take up an even larger percentage of the power consumption of the core.

For an asynchronous reconfigurable processor like DRAP, communication between cells is controlled locally by a mechanism called handshaking. The elimination of the global clock signal makes DRAP inherently more scalable than RICA in terms of power and area overhead.

For dynamically reconfigurable processors like DRAP and RICA, the datapaths mapped on their arrays are of varying lengths. Hence the critical path of each configuration context changes depending on the datapaths being mapped and the level of pipelining required. This complicates the issue of controlling reconfiguration, i.e. knowing when a configuration context has finished computing.

RICA uses the global clock and a special cell in its array to control the amount of time for which each mapped step persists. DRAP, on the other hand, lacks a global

clock and hence has no notion of how much time a configuration context must persist for. A method was needed to control reconfiguration in DRAP: to indicate when the datapaths in a configuration context can start computing and when they have finished and the next step can be loaded. It was important that this method has little or no effect on the scalability of DRAP.

A scheme was devised for DRAP to indicate when a configuration context has terminated and hence when the next one can be loaded. The scheme is novel and can be applied to other asynchronous reconfigurable architectures. A special cell in DRAP interacts with certain designated cells in the array, called the endpoint cells and extracts from their handshaking signals an indicator that the current step has finished. The scheme was designed to be scalable and places no special requirements on the routing algorithm. When moving from a 225 cell array to a 400 cell array, no additional designated endpoint cells needed to be added for correct functionality to occur. The DRAP tool flow was designed to implement the reconfiguration scheme.

9.1.2 Implicit Pipelining

Most mobile applications spend over 95% of execution time in kernel steps, while the remaining time is spent in sequential configurations [4]. The asynchronous operation cells of DRAP inherently contain latches/flip-flops, which are controlled by the handshake signals. As a result, datapaths built with such cells are implicitly pipelined. In most cases this results in full pipelining of the datapaths. This is a particularly useful feature of DRAP because it makes it easy to fully pipeline the kernel steps of mobile applications. There is a class of datapaths for which implicit pipelining does not result in full pipelining. This is referred to as bypass datapaths (Section 7.1.6). For this class, explicit pipelining through asynchronous registers must be added to balance the datapath and hence achieve full pipelining. Asynchronous register cells were distributed in the DRAP array for this purpose and the tool flow was designed to identify bypass datapaths and attempt to balance them and make them fully pipelined.

The ease of pipelining applications on DRAP gives rise to two important properties. The first is an increase in routability over RICA. Because DRAP offers implicit pipelining, it requires a smaller number of interconnect channels to route pipelined applications than RICA. Explicit pipelining on RICA requires connecting additional registers into the datapaths. This increases the pressure on the reconfigurable interconnect. On DRAP, the asynchronous registers used for implicit pipelining are already a hard-wired part of the datapaths. For the bilinear demosaic benchmark, RICA needed a 5-channel interconnect to comfortably route the pipelined application on the array. For DRAP, a 4 channel interconnect was enough to route the same algorithm.

The second property is the ability to achieve high throughputs. DRAP achieved a throughput performance of up to 170x larger than ARM7 and up to 4x larger than the TIC64x VLIW. It also achieved throughputs 1.3x to 2.8x larger than the maximum achieved by RICA for the same applications.

9.1.3 Reduced Program Size

One of the challenges of designing an asynchronous reconfigurable architecture was the design of the interconnect structure. The general interconnect structure of such architectures can be conceptually modelled after that of an equivalent synchronous one. However, to communicate information, asynchronous circuits require more wires than their synchronous counterparts. Additionally, an asynchronous channel connecting a cell to multiple receivers cannot be split or shared without additional complex circuitry to acknowledge every transition on the channel. The traditional method of performing the conditional synchronisation of the acknowledge signals from multiple receivers to a single sender is to use a tree of C-elements and other configurable logic that allow the inputs of a C-element to be ignored or asserted depending on the configuration. One of the disadvantages of the traditional method is the increase in the number of configuration bits over an equivalent interconnect design for synchronous communication. Additional configuration bits are required to

control the conditional synchronisation of the acknowledge signals through controlling C-element tree structure.

A new method for performing conditional communications in programmable asynchronous logic was described in this thesis. It employs select signals that are already used to control the data routing switches, to control the conditional synchronisation of the acknowledge signals. The select signals identify the active acknowledge signals, which are then routed through. The inactive acknowledge signals, where no data will pass through, are preset. As a result of using this newly devised method, there was no increase in the number of configuration bits of the DRAP interconnect when compared to an equivalent interconnect design for synchronous communication.

Certain aspects of the DRAP design allowed a reduction in the program size when compared to a RICA design.

The asynchronous operation cells of DRAP inherently contain latches/flip-flops, which provide ease of pipelining on the array and increase its routability when compared to RICA. As a result, fewer pipelining dedicated register elements than equivalent clocked designs, such as RICA, were needed. The reduction in the number of the distributed register cells and their corresponding configuration bits achieved memory savings on DRAP. For a 15x15 array, DRAP used 71% fewer registers than an equivalent RICA and still achieved full pipelining of the benchmarks mapped on it. This translated to a reduction of 11% in the number of configuration bits for DRAP.

Additionally, DRAP was shown to have a higher routability than RICA. A DRAP array with a smaller number of interconnect channel than an equivalent RICA can run and fully pipeline the same range of applications. A reduction in the number of channels results in a big reduction in the number of configuration bits and hence program size of the array. A 4-channel DRAP array requires 26% less configuration

bits than a 5-channels equivalent RICA array while being able to run the same pipelined applications.

9.1.4 Event-driven Energy Consumption

Asynchronous design techniques implement communication and synchronisation among units at a local level through the use of handshaking. Asynchronous logic inherently implements the synchronous equivalent of fine-grain clock gating. Parts of the array that do not contribute to the computation are automatically turned off and have no switching activity.

The results in this thesis show that DRAP offered a significant reduction in power consumption compared to leading processors. DRAP outperformed ARM7 and the TIC64x VLIW processors by providing 17x – 20x and 25x – 29x less power consumption for the same benchmarks running at the same throughputs, respectively. It consumed 2.3x – 3.4x more power than an equivalent ASIC design of each tested algorithm. Finally, compared to an equivalent RICA based design, DRAP resulted in 1.6x – 1.9x reduction in power consumption when running the same algorithms at the same throughputs.

The decrease in power consumption came at an increase in area of the DRAP array. This is a result of the local control structures associated with asynchronous logic. The area of the DRAP array was 22% larger than that of the equivalent RICA array. However, as mentioned above, the DRAP array resulted in an 11% reduction in program size. Additionally, a 4-channel DRAP array, which is equivalent in functionality to a 5-channel RICA one, is 4.4% larger than its RICA equivalent and requires 26% less program memory. The results of the comparison of DRAP to other architectures are summarised in Table 9.1 and Table 9.2.

Table 9.1: A summary of DRAP vs. RICA and DRAP vs. DSP/VLIW processors.

DRAP vs. RICA	DRAP vs. DSP/VLIW
• Lower power: up to 1.9x • Increased scalability and routability • Less configuration bits (i.e. smaller area for program RAM): up to 26% • Larger area: up to 22% • Easier to pipeline hence higher throughputs more easily achieved: up to 2.8x	• Lower power: up to 20x/29x • Distributed registers as opposed to centralised register file • Distributed data memory access • Higher throughput • Larger program size

Table 9.2: A summary of DRAP vs. ASIC.

DRAP vs. ASIC
• Flexible • Programmable using high-level C language • Higher power: up to 3.4x • Larger area: up to 17x • ASICs should be able to achieve a higher degree of parallelism due to reduced area limits • If DRAP replaces several hardwired Intellectual Properties (IPs), its distributed memory removes the need for a shared bus and hence reduces power

9.2 Future Direction

The work defined in this thesis provides the basis for several avenues of further research. This section presents some of those directions: the ones which could provide the most benefit to the development of DRAP and reconfigurable architectures in general. Future work on DRAP needs to focus on two aspects: the software - where there is room for improvements on the DRAP and RICA tool flow, and the hardware design of DRAP. These are mutually dependent - improvements to the architecture require software to support them, and new ideas in the software can allow for simpler hardware. Therefore, these efforts should proceed in tandem.

9.2.1 Software

A platform is only as good as the tools that support it. To go beyond the realms of pure research, the tools need to be improved in the following ways:

Automation: Certain stages of the DRAP tool flow described in Section 7.2 are currently performed manually. These include routing modifications, and DRAP-specific pipeline optimisations. These are inherently automatable, even using the simple trial and error approach that was followed manually.

Throttling: The current DRAP aims at providing full pipelining for any mapped function and has no ability of controlling its speed and thus its active power consumption. DRAP could control its speed through software, at no extra cost to the hardware. This can be achieved by controlling how much explicit pipelining should be introduced for bypass datapaths.

Enhanced Optimisations: The main thread of future research would concentrate on optimising the scheduling stage to reduce or eliminate bypass datapaths, using improved scheduling and routing algorithms. Explicit pipelining should ideally be performed in tandem with the routing phase, as a constraint in the routing algorithm, to minimise imbalance between stages.

9.2.2 Hardware

The second aspect of future research on DRAP is the hardware design. More specifically, the following areas need further exploration:

Operational cell Design: The DRAP operational cells are heterogeneous coarse-grained cells implemented using 4-phase bundled data asynchronous techniques. Using 4-phase signalling was preferred over 2-phase because it is more widely supported by asynchronous tools (See Section 2.1.3 and Section 3.1). On the other hand, Section 5.2 shows that 4-phase signalling doubles the delay in moving data over 2-phase signalling. There are emerging 2-phase methods [119][120][121][122] which could be tested on DRAP in order to perform a full analysis.

Additionally, the GALS design approach [85] can be tested on the DRAP and RICA family of architectures to provide a third way between a fully synchronous and asynchronous processors. One way of doing this is to create zones within the RICA architecture with each zone containing a variety of operational cells. The local connections within the zones would be synchronous to a local clock and the zones would connect to each other via an asynchronous protocol.

The next generation DRAP should contain controllable pipelining within large operational cells in an effort to improve throughput. It should also have multiple JUMP cells and program counters. This would allow DRAP to run several kernels at once, with each running at its own maximum speed.

Interconnect Design: This thesis examined a small realm of possible interconnect architectures, the island-style one. There is room to explore other interconnect schemes to allow a better scalability of the array and increase routability.

Memory: Further benefits could come from methods for reducing program memory usage, as this constitutes a considerable part of the total power and area of the device.

9.2.3 Analysis

A prototype DRAP was designed and compared to RICA and other architectures by using two benchmarks which reflect mobile requirements. As a next step, larger benchmarks for mobile applications should be simulated on DRAP. The simulation should include both datapaths and memory power results so that a more complete comparison with RICA and the other architectures can be obtained. It should be performed over several process technologies in order to test the effect of moving to lower processes on asynchronous architectures in general and DRAP in particular, to see if the expected trend holds true - that of DRAP's advantage improving as static power consumption becomes ever more dominant.

The prototype DRAP was compared to an equivalent ASIC design of each benchmark. This could allow DRAP to be compared indirectly to other architectures such as FPGAs. There are few detailed comparisons in literature between FPGAs and ASICs. In [123], FPGAs were estimated to be 3.4x to 4.6x slower, 5x to 35x larger and consume 7x to 14x more dynamic power than an equivalent ASIC. As a next step, it would be desirable to compare directly DRAP with FPGAs using benchmarks from mobile applications.

Once a stable model has been completed, the next step would be to get an actual implementation in silicon, or at least a synthesisable soft IP, to be able to test it and provide proof of concept for eventual commercialisation.

References

[1] K. A. Fawaz, T. Arslan, S. Khawam, M. Muir, I. Nousias, I. Lindsay, A. Erdogan, "A Dynamically Reconfigurable Asynchronous Processor for Low Power Applications," Conference on Design and Architectures for Signal and Image Processing (DASIP), October 2010.

[2] K. A. Fawaz, T. Arslan, S. Khawam, M. Muir, I. Nousias, I. Lindsay, A. Erdogan, "A Dynamically Reconfigurable Asynchronous Processor," IEEE 8th Symposium on Application Specific Processors (SASP), June 2010.

[3] Zain-ul-Abdin and B. Svensson, “Evolution in Architectures and Programming Methodologies of Reconfigurable Computing”, Microprocessors and Microsystems, vol. 33, no. 3, pp. 161–178, May 2009.

[4] S. Khawam, I. Nousias, M. Milward, Y. Yi, M. Muir, T. Arslan, “The Reconfigurable Instruction Cell Array,” IEEE Transactions on Very Large Scale Integration (VLSI) Systems, vol. 16, no. 1, 2008.

[5] R. Manohar, “Reconfigurable Asynchronous Logic,” In IEEE Custom Integrated Circuits Conference, CICC '06, pp. 13-20, 2006.

[6] A. Peeters, “The ‘Asynchronous’ Bibliography”. Available at: http://www.win.tue.nl/async-bib/.

[7] J. Garside, “The Asynchronous Logic Homepage”. Available at: http://intranet.cs.man.ac.uk/apt/async/.

[8] “The Advanced Processor Technology Group Publications”. Available at: http://apt.cs.man.ac.uk/publications/.

[9] University of Newcastle upon Tyne, “Asynchronous Group”. Available at: http://async.org.uk/.

[10] "Achronix Semiconductor Corporation". Available at: http://www.achronix.com/.

[11] "TIEMPO". Available at: http://www.tiempo-ic.com/.

[12] "Fulcrum Microsystems". Available at: http://www.fulcrummicro.com/.

[13] A. Davis, S. M. Nowick, "An Introduction to Asynchronous Circuit Design". Technical Report UUCS-97-013, Computer Science Department, University of Utah, Sep. 1997.

[14] C. H. van Berkel, S. M. Nowick et al, "Scanning the Technology: Applications of Asynchronous Circuits". Proceedings of the IEEE, Vol. 87, Issue 2, pp. 223-233; 1999.

[15] J. Sparso, S. Furber, "Principles of Asynchronous Circuit Design: A Systems Perspective". European Low-Power Initiative for Electronic System Design. Kluwer Academic Publishers, ISBN: 0-7923-7613-7, Jan 2002.

[16] W. B. Toms, "Synthesis of Quasi-Delay-Insensitive Datapath Circuits". University of Manchester, Feb 2006.

[17] ACID-WG, "Design Automation and Test for Asynchronous Circuits and Systems". ACID Working Group Report (3rd Edition), 2004.

[18] S. Furber, P. Woods, "Four-phase Micropipeline Latch Control Circuits". IEEE Transactions on VLSI Systems, 4(2): 247-253, June 1996.

[19] S. Furber et al, "A Micropipelined ARM". Proceedings of VLSI 93, pages 5.4.1-5.4.10, September 1993.

[20] D. E. Muller, "Asynchronous Logics and Application to Information Processing". Switching Theory in Space Technology. Howard Aiken and William Mann (Eds.), Stanford U. Press, Stanford, California, 1963.

[21] T. Verhoeff, “Delay-Insensitive Codes – An Overview”. Eindhoven University of Technology, January 1987.

[22] F.-C. Cheng, “Practical Design and Performance Evaluation of Completion Detection Circuits”. Proc. International Conf. Computer Design (ICCD), pp. 354-359, Oct. 1998.

[23] H. Lam; C. Tsui; , "High Performance and Low Power Completion Detection Circuit," Circuits and Systems, 2003. ISCAS '03. Proceedings of the 2003 International Symposium on, vol.5, pp. V-405- V-408 vol.5, 25-28 May 2003.

[24] A. Mokhov, D. Sokolov, A. Yakovlev. “Completion Detection Optimisation”. Technical Report Series: University of Newcastle upon Tyne, October 2005. Available at: http://async.org.uk/tech-reports/.

[25] M.E. Dean, D.L. Dill, M. Horowitz. “Self-Timed Logic Using Current-Sensing Completion Detection (CSCD)”. Journal of Signal Processing, vol. 7, no. 1-2, pp 7-16, pp 269-285, 1994.

[26] H. Lampinen, O. Vainio, “Dynamically Biased Current Sensor-Sensing Completion Detection”. The 2001 IEEE International Symposium on Circuits and Systems, vol. 4, pages: 394-397, 2001.

[27] M. Hevery, “Asynchronous Circuit Completion Detection by Current Sensing”. ASIC/SOC Conference, 1999. Proceedings. Twelfth Annual IEEE International , pp.322-326, 1999.

[28] O.C. Akgun, Y. Leblebici, E.A. Vittoz, “Design of Completion Detection Circuits for Self-timed Systems Operating in Subthreshold Regime”. "Research in Microelectronics and Electronics Conference, 2007. PRIME 2007, pp.241-244, 2-5 July 2007.

[29] E. Grass, R. C. S. Morling, I. Kale “Activity-Monitoring Completion-Detection (AMCD): A New Single Rail Approach to Achieve Self-Timing”. Proc of 2nd International Symposium of Advanced Research in Asynchronous Circuits and Systems, pp 143-149, 1996.

[30] D. A. Huffman, “The Synthesis of Sequential Switching Circuits”, J. Franklin Inst., vol. 257, no 3 pages 161-90, 275-303, March 1954.

[31] R. M. Fuhrer, S. M. Nowick et al, “MINIMALIST: An Environment for the Synthesis, Verification and Testability of Burst-Mode Asynchronous Machines”. Tech. Report #CUCS-020-99, Columbia University, 1999.

[32] D. E. Muller, W. S. Bartky, "A theory of asynchronous circuits". In Proceedings of an International Symposium on the Theory of Switching, pp. 204--243, Harvard University Press, Apr. 1959.

[33] University of Hamburg. “Petri Nets World”. Available at: http://www.informatik.uni-hamburg.de/TGI/PetriNets/.

[34] A. Smirnov, A. Taubin, ”Synthesizing Asynchronous Micropipelines with Design Compiler”. SNUG’06, September 2006.

[35] A. J. Martin, “ The Limitation to Delay-Insensitivity in Asynchronous Circuits”. Advanced Research in VLSI: Proceedings of the Sixth MIT Conference, pages 263-278. MIT Press, 1990.

[36] L. Lavagno, K. Keutzer, A. Sangiovanni-Vincentelli, "Algorithms for Synthesis of Hazard-Free Asynchronous Circuits". In Proc. ACM/IEEE Design Automation Conference, pp. 302--308, IEEE Computer Society Press, 1991.

[37] J. Frenzel, “Asynchronous Design: Ready for Prime Time?”. University of Idaho, 2002.

[38] A. L. Davis, “A Data Driven Machine Architecture Suitable for VLSI Implementation”. Proceeding of Caltech Conference on VLSI, pages 479 – 494, January 1979.

[39] M. B. Josephs, S. M. Nowick,C. H. Van Berkel, “Modeling and Design of Asynchronous Circuits”. Proceedings of the IEEE, Vol.87, Iss.2, Pages: 234-242, Feb 1999.

[40] S. M. Nowick and D. L. Dill, “Synthesis of Asynchronous State Machines Using a Local Clock,” in Proc. ICCAD, 1991, pp. 192–197.

[41] K. Y. Yun, “Synthesis of Asynchronous Controllers for Heterogeneous Systems,” Ph.D. dissertation, Stanford Univ., Stanford, CA, 1994.

[42] A. Davis, B. Coates, K. Stevens, “Automatic synthesis of fast compact self timed control circuits,” in Proc. 1993 IFIP Working Conf. Asynchronous Design Methodologies, Manchester, U.K., pp. 193–207.

[43] R. M. Fuhrer, S. M. Nowick, M. Theobald, N. K. Jha, L. Plana, “MINIMALIST: An Environment for the Synthesis and Verification of Burst-mode Asynchronous Machines,” in Proc. IEEE/ACM Int. Workshop Logic Synthesis, 1998.

[44] L. Lavagno, A. Sangiovanni-Vincentelli. “Algorithms for Synthesis and Testing of Asynchronous Circuits.” Kluwer Academic Publishers, 1993.

[45] J. Cortadella. “Petrify: A Tutorial for the Designer of Asynchronous circuits”. Available as part of the petrify tool package from: http://www.lsi.upc.es/jordic/petrify.

[46] J. Cortadella. “PETRIFY: A Tool for Manipulating Concurrent Specifications and Synthesis of Asynchronous Controllers”. IEICE Trans. Inform. Syst., vol. E80-D, no. 3, pp. 315 -325, 1997.

[47] A.J. Martin. Programming in VLSI: From communicating processes to delay-insensitive circuits. In C.A.R. Hoare, editor, Developments in Concurrency and Communication, pages 1 - 64. Addison-Wesley, Reading, MA, 1990.

[48] A. Peeters, M. De Wit, "Asynchronous Circuit Design Using Handshake Solutions," SOC Conference, 2008 IEEE International , vol., no., pp.391-392, 17-20 Sept. 2008.

[49] K. van Berkel, “Handshake Circuits: An Asynchronous Architecture for VLSI Programming,” International Series on Parallel Computation, vol. 5. Cambridge, U.K.: Cambridge Univ. Press, 1993.

[50] A. Bardsley, D.A. Edwards. “The Balsa Asynchronous Circuit Synthesis System". Forum on Design Languages, 2000.

[51] J. Lee, S. Lee, K. Cho. “Asynchronous ARM Processor Employing an Adaptive Pipeline Architecture”. In Proceedings of the 3rd international conference on Reconfigurable computing: architectures, tools and applications (ARC'07), 2007.

[52] R. Manohar, A.J. Martin, “Slack Elasticity in Concurrent Computing,” Proc. Int’l Conf. Math. of Program Construction, 1998.

[53] M. Roncken, “Defect-oriented Testability for Asynchronous ICs”. Proceedings of the IEEE, vol.87, no.2, pp.363-375, Feb 1999.

[54] D.P. Vasudevan, A. Efthymiou, "A Partial Scan Based Test Generation for Asynchronous Circuits," 11th IEEE Workshop on Design and Diagnostics of Electronic Circuits and Systems, 2008 - DDECS 2008, pp.1-4, 16-18 April 2008.

[55] J. G. Andrews, A. Ghosh, R. Muhamed, Fundamentals of WiMAX, Prentice Hall, 2007.

[56] C. Johnson, "LTE in BULLETS", CreateSpace, 2010, ISBN 978-1452834641.

[57] H. Ekstrom, A. Furuskar, J. Karlsson, et al., "Technical solutions for the 3G long-term evolution," Communications Magazine, IEEE, vol.44, no.3, pp. 38- 45, March 2006.

[58] O. Silven, T. Rintaluoma, K. Jyrkkä, Implementing energy efficient embedded multimedia, Proceedings of SPIE - Volume 6074, Feb 10, 2006.

[59] A.F. Harris, R. Snader, R. Kravets, “Mobile system energy conservation for adaptive multimedia applications,” In Proceedings of the 2008 International Symposium on a World of Wireless, Mobile and Multimedia Networks (WOWMOM '08). IEEE Computer Society, Washington, DC, USA, 1-11.

[60] P. Heysters, G. Smit, E. Molenkamp, “A Flexible and Energy-Efficient Coarse-Grained Reconfigurable Architecture for Mobile Systems,” In J. Supercomput, vol.26, no.3, pp.283-308, November 2003.

[61] “ITU”. Available at http://www.itu.int/.

[62] Framework and overall objectives of the future development of IMT-2000 and systems beyond IMT-2000 (www.ieee802.org/secmail/pdf00204.pdf).

[63] M. Woh, S. Mahlke, T. Mudge, C. Chakrabarti, "Mobile Supercomputers for the Next-Generation Cell Phone," Computer , vol.43, no.1, pp.81-85, Jan. 2010.

[64] J.V. Woods, P. Day, S.B. Furber, J.D. Garside, N.C. Paver, S. Temple, “AMULET1 : An Asynchronous ARM Microprocessor”, IEEE Transactions on Computers, vol. 46, no. 4, April 1997.

[65] S.B. Furber, J.D. Garside, S. Temple, J. Liu, P. Day, N.C. Paver, « Amulet 2 e : An Asynchronous Embedded Controller », Proceedings Async'97, pp. 290-299, IEEE Computer Society Press, April 97.

[66] J.D. Garside et al., ''AMULET3i—An Asynchronous System-on-Chip,'' Proc. Int'l Symp. Advanced Research in Asynchronous Circuits and Systems (ASYNC 00) , pp. 162-175, IEEE CS Press, 2000.

[67] A. Bink, R. York. 2007. "ARM996HS: The First Licensable, Clockless 32-Bit Processor Core". Journal of IEEE Micro, pp. 58-68, March 2007.

[68] R. Hartenstein, "Coarse Grain Reconfigurable Architectures". In Asia and South Pacific Design Automation Conference (ASP-DAC), pages 564 - 569, Yokohama, Japan, 30 January - 2 February 2001.

[69] D. Lewis et al., "The Stratix II logic and routing architecture," in FPGA '05: Proceedings of the 2005 ACM/SIGDA 13th international symposium on Field-programmable gate arrays, pp. 14–20, New York, NY, USA: ACM, 2005.

[70] L. Shang, A. S. Kaviani, K. Bathala, "Dynamic power consumption in Virtex-II FPGA family," in FPGA '02: Proceedings of the 2002 ACM/SIGDA tenth international symposium on Field-programmable gate arrays, pp. 157–164, New York, NY, USA: ACM, 2002.

[71] A. Gayasen, N. Vijaykrishnan, M. Irwin, "Exploring technology alternatives for nano-scale FPGA interconnects," June 2005.

[72] S. Khawam, "Domain-specific and Reconfigurable Instruction Cells based Architectures for Low-Power SoC", PhD Thesis, University of Edinburgh, April 2006.

[73] M. Muir, "Re-Targetable Tools and Methodologies for the Efficient Deployment of High-Level Source Code on Coarse-Grained Dynamically

Reconfigurable Architectures", PhD Thesis, University of Edinburgh, October 2009.

[74] I. Nousias, "Reconfigurable Computing: The Reconfigurable Instruction Cell Array: Reconfiguration and Interconnects", PhD Thesis, University of Edinburgh, April 2009.

[75] S. Hauck, S. Burns, G. Borriello, C. Ebeling, "An FPGA for Implementing Asynchronous Circuits," IEEE Design and Test of Computers, vol. 11, no. 3, pp. 60-69, 1994.

[76] K. Maheswaran, "Implementing Self-Timed Circuits in Field Programmable Gate Arrays," master's thesis, Univ. of California Davis, 1995.

[77] R. Payne, "Asynchronous FPGA Architectures," IEE Computers and Digital Techniques, vol. 143, no. 5, 1996.

[78] R. U. R. Mocho, G. H. Sartori, R. P. Ribas, A. I. Reis, "Asynchronous Circuit Design on Reconfigurable Devices," Proceedings of the 19th annual symposium on Integrated circuits and systems design, Ouro Preto, MG, Brazil, pp. 20-25, 2006.

[79] Y. Zafar, M. Ahmed, "A Novel FPGA Compliant Micropipeline", IEEE Trans. on CAS II – Express Briefs, Vol 52, No 9, pp. 611-615, September 2005.

[80] Q. T. Ho, J. B Rigaud, L. Fesquet, M. Renaudin, R. Rolland, "Implementing Asynchronous Circuits on LUT Based FPGAs," Proc. Int'l Conf. Field Programmable Logic and Applications, 2002.

[81] N. Huot, H. Dubreuil, L. Fesquet, M. Renaudin, "FPGA Architecture for Multi-style Asynchronous Logic", DATE 2005, pp. 32-33, 2005.

[82] D.L. How, “A Self Clocked FPGA for General Purpose Logic Emulation,” Proc. IEEE Custom Integrated Circuits Conf., 1996.

[83] C. Traver, R.B. Reese, M.A. Thornton, “Cell Designs for Self-Timed FPGAs,” Proc. ASIC/SOC Conf., 2001.

[84] X. Jia, J. Rajagopalan, R. Vemuri, “A Dynamically Reconfigurable Asynchronous FPGA Architecture,” Field Programmable Logic and Application, 14th International Conference , FPL 2004, Leuven, Belgium, Proceedings, vol. 3203, pp. 836-841, 2004.

[85] A. Royal, P. Y. K. Cheung, “Globally Asynchronous Locally Synchronous FPGA Architectures,” Field Programmable Logic and Application, 13th International Conference, FPL 2003, Lisbon, Portugal, Proceedings (Lecture Notes in Computer Science), vol. 2778, pp. 355-364, 2003.

[86] J. Teifel, R.Manohar, “An Asynchronous Dataflow FPGA Architecture,” IEEE Transactions on Computers, vol. 53, no. 11, pp. 1376–1392, 2004.

[87] C. G. Wong, A. J. Martin, P. Thomas, “An Architecture for Asynchronous FPGAs”, In IEEE International Conference on Field-Programmable Technology, 2003.

[88] J. Rose, S. Brown, “Flexibility of Interconnection Structures for Field-programmable Gate Arrays,” IEEE Journal of Solid-State Circuits, vol. 26, no. 3, pp. 277-282, 1990.

[89] B. Ghavami, M. Mirza-Aghatabar, H. Pedram, S. Hessabi, “Analysis and Fast Estimation of Energy Consumption in Template Based QDI Asynchronous Circuits,” International Symposium on Integrated Circuits, ISIC '07, pp. 445-448, 2007.

[90] K. Sun, X. Pan, J. Wang, “Design of a Novel Asynchronous Reconfigurable Architecture for Cryptographic Applications,” IEEE International Multi-

Symposiums on Computer and Computational Sciences, IMSCCS 2006, pp. 751-757, 2006.

[91] M. Mishra, T. J. Callahan, T. Chelcea, G. Venkataramani, M. Budiu, S. C. Goldstein, “Tartan: Evaluating spatial computation for whole program”, ASPLOS’06, October 2006.

[92] S. C. Goldstein, H. Schmit, et al. PipeRench: a coprocessor for streaming multimedia acceleration. In International Symposium on Computer Architecture (ISCA), pages 28–39, May 1999.

[93] T. Bjerregaard, J. Sparsø, “A router architecture for connection-oriented service guarantees in the MANGO clockless network-on-chip”, In Proceedings of the Design, Automation and Test in Europe Conference and Exhibitions (DATE’05), Vol. 2, , pp. 1226-1231, March 2005.

[94] X. Li, B. K. Gunturk, L. Zhang, "Image demosaicing: A systematic survey," in Proc. SPIE-IS&T Electronic Imaging, Visual Communications and Image Processing, Jan. 2008.

[95] A. Efthymiou, J.D. Garside, "Adaptive pipeline structures for speculation control," Asynchronous Circuits and Systems, 2003. Proceedings. Ninth International Symposium, pp. 46- 55, 12-15 May 2003.

[96] S. Khawam, T. Arslan; “Switch-box design for synthesizable coarse-grain arrays for system-on-chip applications”, In Proceedings of 2004 IEEE International Conference on Field-Programmable Technology (FPT), pp.465 – 468, 2004.

[97] S. Khawam, T. Arslan, F. Westall; “Unidirectional switch-boxes for synthesizable reconfigurable arrays”, 12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM 2004) pp. 293 – 295, 20-23 April 2004.

[98] Jing Huang, M.B. Tahoori, F. Lombardi, "Routability and fault tolerance of FPGA interconnect architectures," Test Conference, 2004. Proceedings. ITC 2004. International , vol., no., pp. 479- 488, 26-28 Oct. 2004.

[99] P. Christie, D. Stroobandt, “The Interpretation and Application of Rent's Rule”, IEEE Trans. on VLSI Systems, Special Issue on System-Level Interconnect Prediction, vol. 8, no. 6, pp. 639-648, 2000.

[100] M.I. Masud, S.J.E. Wilton, “A New Switch Block for Segmented FPGAs'”, in International Workshop on Field Programmable Logic and Applications , Aug. 1999.

[101] D. Harris, S. Harris, “Digital Design and Computer Architecture”. Morgan Kaufmann, ISBN: 978-0123704979, March 2007.

[102] J. W. Cooley, J. W. Tukey, “An algorithm for the machine calculation of complex Fourier series,” Math Comput. 19, pp. 297-301, 1965.

[103] GNU, Boston, MA, "GNU C compiler," 2005 [Online]. Available: http://gcc.gnu.org/.

[104] Y. Yi , I. Nousias , M. Milward , S. Khawam , T. Arslan , I. Lindsay, System-level scheduling on instruction cell based reconfigurable systems, Proceedings of the conference on Design, automation and test in Europe: Proceedings, March 06-10, 2006, Munich, Germany.

[105] V. Betz , J. Rose, “VPR: A new packing, placement and routing tool for FPGA research,” Proceedings of the 7th International Workshop on Field-Programmable Logic and Applications, p.213-222, September 01-03, 1997.

[106] B. E. Bayer. Color imaging array, July 1976. U.S. Patent 3971065.

[107] M. McGuire, W. College, “Efficient, High-Quality Bayer Demosaic Filtering on GPUs,” Submitted to Journal of Graphics Tools, 2009.

[108] W. Yu, “Colour demosaicking method using adaptive cubic convolution interpolation with sequential averaging,” VISP, 153(5):666–676, October 2006.

[109] R. Woods, R. Gonzalez, S. Eddins. Digital Image Processing Using MATLAB. Prentice-Hall, 2004.

[110] C. E. Duchon, “Lanczos filtering in one and two dimensions,” Journal of Applied Meteorology, vol. 18, no. 8, pp. 1016–1022, 1979.

[111] Digital Video Broadcasting (DVB); Framing Structure, Channel Coding and Modulation for Satellite Services to Handheld Devices (SH) below 3 GHz, ETSI EN 302 583.

[112] DVB–SH Implementation Guidelines, ETSI A12.

[113] C.C. Wang, J.M. Huang, H.C. Cheng, “A 2K/8K Mode Small- Area FFT Processor for OFDM Demodulation of DVB-T Receivers”, IEEE Transactions on Consumer Electronics, Vol. 51, Issue 1,pp. 28-32, Feb. 2005.

[114] M. Vetterli and H. J. Nussbaumer, "Simple FFT and DCT algorithms with reduced number of operations," Signal Processing 6 (4), 267–278 (1984).

[115] ARM Ltd., Cambridge, U.K., "ARM7 thumb family datasheet," ARM DOI 0035-3/02.02, 2002.

[116] S. Agarwala et al., "A 600-MHz VLIW DSP," IEEE J. Solid-State Circuits , vol. 37, no. 11, pp. 1532-1544, Nov. 2002.

[117] “Texas Instruments”. Available at: http://www.ti.com/.

[118] G. Martinez, "TI TMS320VC5501/02 power consumption summary," Appl. Rep. SPRAA48, 2004.

[119] C. LaFrieda, B. Hill, R. Manohar, "An Asynchronous FPGA with Two-Phase Enable-Scaled Routing," IEEE Symposium on Asynchronous Circuits and Systems (ASYNC), pp.141-150, 3-6 May 2010.

[120] A. Peeters, F. te Beest, M. de Wit, W. Mallon, "Click Elements: An Implementation Style for Data-Driven Compilation," IEEE Symposium on Asynchronous Circuits and Systems (ASYNC), pp.3-14, 3-6 May 2010.

[121] A. Mitra, W. McLaughlin, S. Nowick, "Efficient Asynchronous Protocol Converters for Two-Phase Delay-Insensitive Global Communication," IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC), pp.186-195, 12-14 March 2007.

[122] Hock Soon Low, Delong Shang, Fei Xia, A. Yakovlev, "Variation Tolerant AFPGA Architecture," IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC), pp.77-86, 27-29 April 2011.

[123] I. Kuon, J. Rose, "Measuring the Gap Between FPGAs and ASICs," IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol.26, no.2, pp.203-215, Feb. 2007.

Appendix A

DRAP Cells and their Operations

This appendix provides more information on the DRAP operational cells. For each cell, the commands that were implemented are described. A 32-bit and an 18-bit version of each of the operational cell were designed for the DRAP evaluation. Unless otherwise specified, the width of the data operated on is the entire available width (32bit/18bit).

A.1 ADDCOMP

Configuration bits: 4 bits

Description of Commands:

- Addition
- Subtraction
- Absolute Difference
- Average
- Equal

- Not Equal
- Greater than or equal (unsigned/signed)
- Greater than (unsigned/signed)
- Less than (signed)
- Less than or equal (signed)

Comments:

- Initially, ADD and COMP were designed as separate cells. For the evaluations, only the combined ADDCOMP was used

A.2 JUMP

Configuration bits: 7 bits

Description of Commands:

- Evaluate flag and go to the next step
- Output address that would occur if the jump is not executed
- Output a pulse START_to_delay signal for non-kernel step
- Keep START_to_delay signal high (until end of kernel) when next step is a kernel

Comments:

- Can jump to a relative or absolute address
- Flag is given two bits and covers several conditions
- The configuration decides what type of start signal to output
- Receives a START signal that tells it when to start working

A.3 LOGIC

Configuration bits: 5 bits

Description of Commands:

- Negation
- OR/NOR
- AND/NAND
- XOR
- Inverse
- Absolute
- Bit reverse
- Sign extend (HI/QI)
- Zero extend (HI/QI)

Comments:

- HI: Half integer
- QI: Quarter integer

A.4 MUL

Configuration bits: 3 bits

Description of Commands:

- Multiplication (signed/unsigned)

Comments:

- Configuration includes space for half integer and quarter integer modes

A.5 REG

Configuration bits: 2/3 bits

Description of Commands:

- Output stored data
- Input new data
- Output stored data then input new data
- Disable
- Input new data then output it (optional)
- Output from memory (optional)

Comments:

- Receives a START signal that tells it when to start working
- For the evaluation of DRAP, the disable command was part of the 2-bit configuration. In cases where, the optional commands are needed, an additional configuration bit is required

A.6 SBUF

Configuration bits: 4 bits

Description of Commands:

- Disable
- Stream Read
- Stream Write
- Stream Read and Write
- Set Read
- Set Write
- Set Read and Write

Comments:

- Receives a START signal that tells it when to start working

A.7 SHIFT

Configuration bits: 5 bits

Description of Commands:

- Shift left/right logical
- Shift right arithmetic
- Shift left/right 15-1
- Shift left/right 14-2
- Shift left/right 12-4
- Shift left/right 10-6
- Shift left/right 8-8

Comments:

- Configuration includes space for half integer and quarter integer modes

A.8 SINK

Configuration bits: 1 bit

Description of Commands:

- Read
- Disable

Comments:

- Receives a START signal that tells it when to start working
- There might be a need to include a counter within this cell. In that case, additional configuration bits are required to set the starting point of the counter

A.9 SOURCE

Configuration bits: 3 bits

Description of Commands:

- Disable
- Write

Comments:

- Receives a START signal that tells it when to start working

Appendix B

The DVB-SH Standard

The DVB-SH standard defines four FFT modes for the OFDM receiver. The numerical values for the 8k, 4k, 2k and 1k modes (for an 8MHz channel) are given in Tables B.1 and B.2 [111]. The elementary period T is 7/64 μs for 8 MHz channels. As can be seen from the table, the duration for the 8K mode with a guard of 1/32 is 924μs.

Table B.1: Numerical values for OFDM parameters (all modes for 8 MHz channels) [111].

Parameter	8k mode	2k mode
Number of carriers K	6 817	1 705
Value of carrier number K_{min}	0	0
Value of carrier number K_{max}	6 816	1 704
Duration T_U (see note 2)	***896 µs***	***224 µs***
Carrier spacing $1/T_U$ (see note)	***1 116 Hz***	***4 464 Hz***
Spacing between carriers K_{min} and K_{max} $(K-1)/T_U$	***7,61 MHz***	***7,61 MHz***

Parameter	4k mode	1k mode
Number of carriers K	3 409	853
Value of carrier number K_{min}	0	0
Value of carrier number K_{max}	3 408	852
Duration T_U (see note 2)	***448 µs***	***112 µs***
Carrier spacing $1/T_U$ (see note)	***2 232 Hz***	***8 929 Hz***
Spacing between carriers K_{min} and K_{max} $(K-1)/T_U$	***7,61 MHz***	***7,61 MHz***
NOTE: Values in bold and italics are approximate values.		

Table B.2: Duration of symbol part for the allowed guard intervals (8 MHz channels) [111].

Mode	8k mode				2k mode			
Guard interval Δ / T_U	1/4	1/8	1/16	1/32	1/4	1/8	1/16	1/32
Duration of symbol part T_U	8 192 × T 896 µs				2 048 × T 224 µs			
Duration of guard interval Δ	2 048 × T 224 µs	1 024 × T 112 µs	512 × T 56 µs	256 × T 28 µs	512 × T 56 µs	256 × T 28 µs	128 × T 14 µs	64 × T 7 µs
Symbol duration $T_S = \Delta + T_U$	10 240 × T 1 120 µs	9 216 × T 1 008 µs	8 704 × T 952 µs	8 448 × T 924 µs	2 560 × T 280 µs	2 304 × T 252 µs	2 176 × T 238 µs	2 112 × T 231 µs

Mode	4k mode				1k mode			
Guard interval Δ / T_U	1/4	1/8	1/16	1/32	1/4	1/8	1/16	1/32
Duration of symbol part T_U	4 096 × T 448 µs				1 024 × T 112 µs			
Duration of guard interval Δ	1 024 × T 112 µs	512 × T 56 µs	256 × T 28 µs	128 × T 14 µs	256 × T 28 µs	128 × T 14 µs	64 × T 7 µs	32 × T 3.5 µs
Symbol duration $T_S = \Delta + T_U$	5 120 × T 560 µs	4 608 × T 504 µs	4 352 × T 476 µs	4 224 × T 462 µs	1 280 × T 140 µs	1 152 × T 126 µs	1 088 × T 119 µs	1 056 × T 115,5 µs

Appendix C

Haste and the TiDE Tool Flow

C.1 Overview

The tool used for designing the asynchronous cells in DRAP was the TiDE design environment from Handshake Solutions. TiDE uses a CSP-like programming language called Haste. [48]

TiDE works as follows (Figure C.1): The circuits are described in Haste and then synthesised in two stages to a Verilog netlist based on cells from a standard-cell library. The first stage translates the Haste code into an intermediate Handshake Circuit in a transparent, syntax-directed process and the next stage maps the Handshake Circuit to a structural Verilog netlist for initial circuit-level optimisation [48]. TiDE has an option that allows the import of non-Haste logic functions and integrates them into the Verilog netlist to produce a complete one. Standard Electronic Design Automation (EDA) tools can then be used to obtain an optimised Verilog netlist.

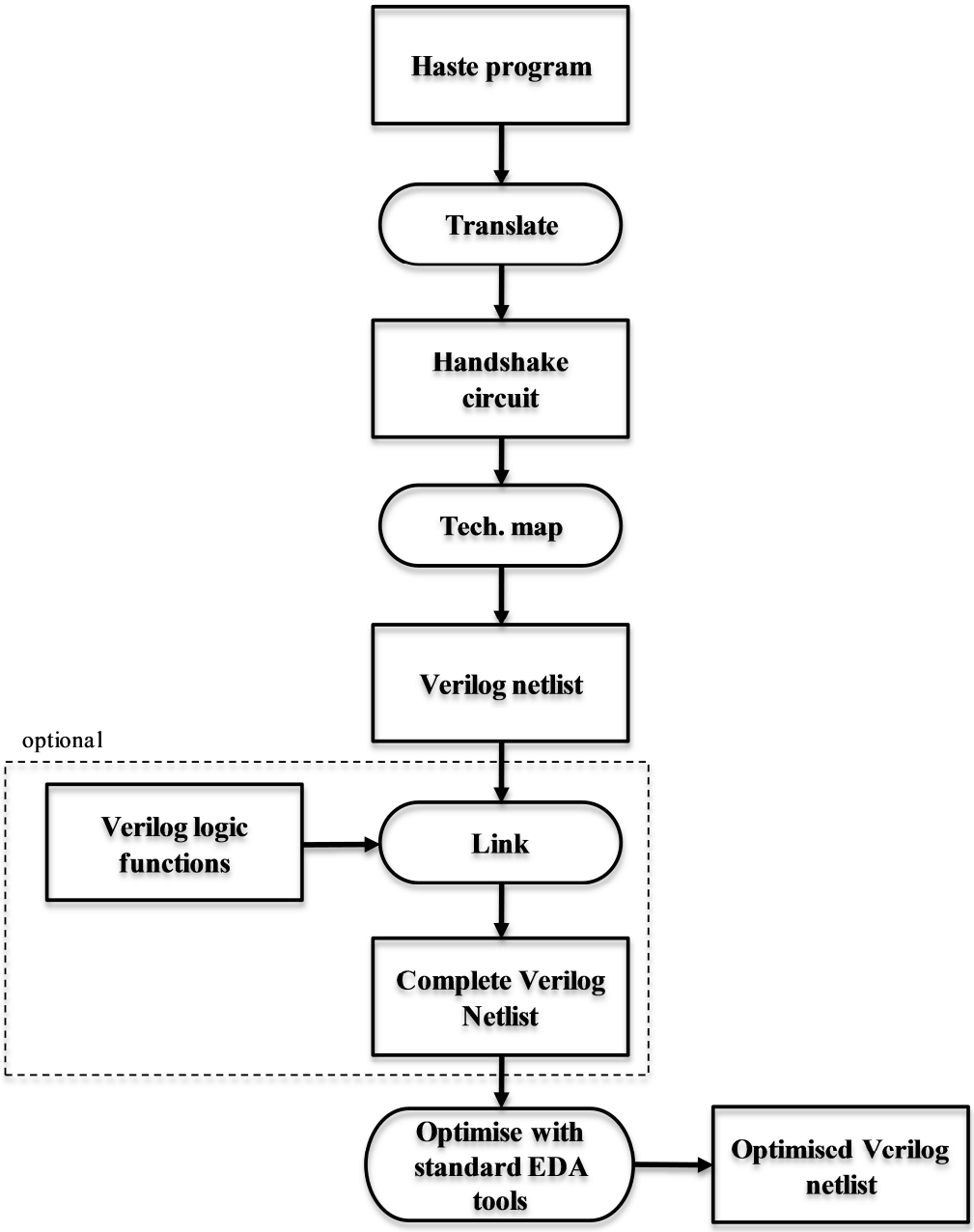


Figure C.1:The TiDE design flow.

C.2 Haste Notations

The following is a description of some the Haste notations used in this thesis:

C.2.1 Basic statements

- x ≔ a : assign the value of a to x
- A?a : Assign the value on the input channel A to the variable a
- B!b : Output the value of b onto channel B

C.2.2 Composition statements

- X ; Y: Execute statement(s) X then execute Y in series
- X || Y: Execute statement(s) X and Y in parallel

Appendix D

C Source Codes

D.1 Bilinear Demosaic – C Source Code

The following is the C source code for the bilinear demosaic filter used as a benchmark on DRAP and other architectures.

```
/*************************************************************
 * Bilinear_Demosaic.c
 *
 * Sami Khawam 2008, Khodor Fawaz 2010.
 *
 * Bilinear_Demosaic. Uses a 3x3 kernel.
 *
 *  -  16-bit input.
 *  -  X,Y size from SBUF
 *
 *************************************************************/

/* Size of image sequence 'source.pnm' */
#define INPUT_NUM_COLUMNS_source  2056
#define INPUT_NUM_ROWS_source     25
#define INPUT_NUM_FRAMES_source     1

// Include this to get the shared objects cache
// struct and update functions.
//#include "shared_regs.h"

#define NO_AMBA
#define NO_SYNC
#define NO_BOX
```

```
#define NUM_LINES   3   // 3 lines of raw input image are required before an RGB output pixel can be calculated


#include <rica/rica.h>
#include <rica/source_sink.h>       // for read_source(), write_sink()
#include <rica/stream_buffers.h>
#include <rica/isp.h>  // for pixel_t, pixel_yuv_t
#include <rica/sregf.h>
#include <rica/irq.h>
#include <stdio.h>
#include <stdlib.h>


inline __attribute__ ((always_inline)) void feed_block_3x3( unsigned int b[3][3], unsigned int c[3]);
inline __attribute__ ((always_inline)) void Bilinear_Demosaic(pixel_t *output_pixel,
                                unsigned int dm_block[3][3], int odd_col, int even_row,
                                int clipping_val);
inline __attribute__ ((always_inline)) void rotate_line_delays();

enum e_yuv_formats {YUV_RGB_PEAK,
                    YUV_RGB_UNPEAK,
                    YUV422_121,
                    YUV422_11,
                    YUV_BAYER_RS,
                    YUV_BAYER_LS};


inline __attribute__ ((always_inline))  void rgb2yuv (
   /* output */
   pixel_yuv_t* output,

   /* input */
   pixel_t input,

   /* input */
   enum e_yuv_formats yuv_format,

   int yuv_y_cof00,
   int yuv_y_cof01,
   int yuv_y_cof02,
   int yuv_yfloor,

   int yuv_cb_cof10,
   int yuv_cb_cof11,
   int yuv_cb_cof12,
   int yuv_cbfloor,

   int yuv_cr_cof20,
   int yuv_cr_cof21,
   int yuv_cr_cof22,
   int yuv_crfloor,

   int yuv_ceiling,
   int yuv_data_min_clip_en,
   int yuv_chroma_high_clip_en
);

inline __attribute__ ((always_inline))  void sub_sample_yuv422
(
   /* input */
   pixel_yuv_t  input,

   /* output */
   pixel_yuv_t* output,

   /* in-out*/
   pixel_yuv_t* delayed_yuv_1,
   pixel_yuv_t* delayed_yuv_2,
```

```
    int current_x,

    enum e_yuv_formats yuv_format

);

int main (void)
{
    int x_size;// = INPUT_NUM_COLUMNS_source;  // the number of pixels per row/line of image
    int y_size;// = INPUT_NUM_ROWS_source;     // the number of rows per image (in pixels)

    int i, j, prev_j;
    int input_pixel;           // input pixel from the source cell
    pixel_t output_RGB;        // output 24bit RGB value
    unsigned int block[3][3];        // 3x3 Window
    unsigned int new_column[3];         // Latest column of pixels
    const int clipping_val = 0xffff;
    int output_pixel;

        pixel_yuv_t   delayed_yuv_1 = {0,0,0};
        pixel_yuv_t   delayed_yuv_2 = {0,0,0};
        pixel_yuv_t   yuv_pixel;
        pixel_yuv_t   final_yuv_pixel;

    // Values going to RGB2YUV
    // Fixed-point at 255
    int yuv_y_cof00     = 0x42;
    int yuv_y_cof01     = 0x7F;
    int yuv_y_cof02     = 0x1D;
    int yuv_yfloor      = 0x1D;
    int yuv_cb_cof10    = -38; //0x7DA;    // -0.14713
    int yuv_cb_cof11    = -74; //0x7B6;    // -0.28886
    int yuv_cb_cof12    = 0x070;
    int yuv_cbfloor     = 0x080;
    int yuv_cr_cof20    = 0x070;
    int yuv_cr_cof21    = -94; //0x7A2;    // -0.51499
    int yuv_cr_cof22    = -18; //0x7EE;    // -0.10001
    int yuv_crfloor     = 0x80;
    int yuv_ceiling     = 0;
    int yuv_data_min_clip_en        = 0;
    int yuv_chroma_high_clip_en     = 0;
    enum e_yuv_formats yuv_format   = YUV422_121;

    int output_en, hsync_en, vsync_en;

    // Give an output with H-Sync and V-Sync low
    #ifndef NO_SYNC
    write_sink_w_sync(0, 0, 0, 0, 1);
    #endif


    #ifndef NO_AMBA
    // Use bit 3 to send an interrupt that says RICA started
    ASM_IRQ_AMBA_WRITE(0x8);

    // Wait for LEON to load the values in SREGFs and SBUFB
    ASM_IRQ_AMBA_WAIT_FOR(0x01);

    //x_size = sregf_read(0, 0);      // Location 0 in SREGF[0]
    //y_size = sregf_read(1, 0);      // Location 0 in SREGF[1]

    //SharedObjectsCache sregf_objects = shared_objects_cache_update();
    #else
    x_size = INPUT_NUM_COLUMNS_source;
    y_size = INPUT_NUM_ROWS_source;
    #endif

    while(1) {
```

```
for(i=0; i<y_size; i++)
{
   rotate_line_delays();

   for(j=0; j<x_size; j++)
   // Trying with decrementing while to see if we can speed up things
   //j = x_size;//-1;
   //do
   {
    //   PIPELINE;              // This creates an asm comment which tells the scheduler to pipeline the kernel
                        GUESS_DEAD_KERNEL_REGISTERS;

      // read in the next incoming raw stream pixel
      input_pixel = read_source(0, true/*enable*/);

      new_column[0] = sbuf_read(0, true/*enable*/);
      new_column[1] = sbuf_read(1, true/*enable*/);
      new_column[2] = input_pixel;

      sbuf_write(0, input_pixel, true/*enable*/);

      feed_block_3x3(block, new_column);

      //output_pixel = input_pixel ;
      Bilinear_Demosaic(&output_RGB, block, (j & 1)^1, (i & 1)^1, clipping_val);   // '1' is for Red line
      // If we started j from an odd number, we would have to remove the ^1

      // Add Box
      #ifndef NO_BOX
      int enable_box = (sregf_objects.box_x_size != 0) &
                  (j >= sregf_objects.box_x_start) & (j <= sregf_objects.box_x_start+sregf_objects.box_x_size) &
                  (i >= sregf_objects.box_y_start) & (i <= sregf_objects.box_y_start+sregf_objects.box_y_size);
      output_RGB.red   = enable_box ? box_red : output_RGB.red;
      output_RGB.green = enable_box ? box_green: output_RGB.green;
      output_RGB.blue  = enable_box ? box_blue : output_RGB.blue;
      #endif

      ASM_COMMENT ("\n\t// RGB 2 YUV");
      rgb2yuv
      (
         &yuv_pixel,
         output_RGB,
         yuv_format,
         yuv_y_cof00,
         yuv_y_cof01,
         yuv_y_cof02,
         yuv_yfloor,
         yuv_cb_cof10,
         yuv_cb_cof11,
         yuv_cb_cof12,
         yuv_cbfloor,
         yuv_cr_cof20,
         yuv_cr_cof21,
         yuv_cr_cof22,
         yuv_crfloor,
         yuv_ceiling,
         yuv_data_min_clip_en,
         yuv_chroma_high_clip_en
      );


      // YUV422 Subsampling
      ASM_COMMENT ("\n\t// YUV422 Sub-Sampling");
      sub_sample_yuv422
      (
         yuv_pixel,
         &final_yuv_pixel,
         &delayed_yuv_1,
```

```
            &delayed_yuv_2,
            (j & 1)^1,              // If we started j from an odd number, we would have to remove the ^1
            yuv_format
          );


          // Output pixel
          //unsigned int v_or_u = (j & 1) ? final_yuv_pixel.u : final_yuv_pixel.v ;
          unsigned int v_or_u = (j & 1) ? final_yuv_pixel.v : final_yuv_pixel.u ;
          // If we started j from an odd number, we would have to swap counter
          //unsigned int v_or_u = (j & 1) ? final_yuv_pixel.u : final_yuv_pixel.v ;


          output_en = 1;

          hsync_en = output_en;
              volatile int volatile_zero = 0;
          vsync_en = output_en & (i==volatile_zero); //<6);       // Only the firstline


          //output_16_bit(vsync_en << 17 | hsync_en << 16 | (final_yuv_pixel.y << 8) | v_or_u, 1);
              //write_sink_w_sync(0, (final_yuv_pixel.y << 8) | v_or_u, hsync_en, vsync_en, 1);          //UYVY
              write_sink_w_sync(0, (v_or_u << 8) | final_yuv_pixel.y, hsync_en, vsync_en, 1);          //YUY2

      //prev_j = j;
      //j--;
      }
      //while (--j);// Try to reduce JUMP loop
      //while (prev_j);// Try to reduce JUMP loop

      // Give an output with H-Sync low
      #ifndef NO_SYNC
      for(j=0; j<2; j++)                // Output several pixels to simulate an interline delay
       write_sink_w_sync(0, 0, 0, vsync_en, 1);
      #endif
    }
    }

    return 0;
}


#define CLIPPING(VALUE,LIMIT)   ({ \
              int t; \
              VALUE = (VALUE <= 0) ? 0: VALUE; \
              VALUE = (VALUE > LIMIT) ? LIMIT : VALUE; })


inline __attribute__ ((always_inline)) void Bilinear_Demosaic(
        pixel_t *output_pixel, unsigned int pixels[3][3],
        int odd_col, int odd_row, int clipping_val)
{

    // The different averages
    unsigned int cross_av;
    unsigned int x_av;
    unsigned int vert_av;
    unsigned int hori_av;

    unsigned int r_odd, r_even;
    unsigned int g_odd, g_even;
    unsigned int b_odd, b_even;
    int r;
    int g;
    int b;

    // The center pixel is pixels[1][1].
    //
    // Calculate the different averages, and then choose the correct one
```

```
#define FORCE_AVG
#ifdef FORCE_AVG
// When having multiple CONST combined, the compiler cannot detect the AVG anymore
#define ASM_2P(CELL_NAME,CONFIG,OUT,IN2,IN1)\
asm\
(\
   #CELL_NAME "\tout= %0 \tin1= %1 \tin2= %2 \tconf= `" # CONFIG\
   : "=r" (OUT)\
   :   "r" (IN1), "r" (IN2)\
)

        #ifdef RICA
#define AVG(O,X,Y)  ASM_2P(ADDCOMP,ADDCOMP_AVG_SI,O,X,Y)
        #else
        #define AVG(O,X,Y)  ((O)=((X)+(Y))/2)
        #endif

unsigned int tmp1, tmp2;
AVG(vert_av, pixels[0][1], pixels[2][1]);
AVG(hori_av, pixels[1][0], pixels[1][2]);
AVG(cross_av, vert_av, hori_av);
AVG(tmp1, pixels[0][0], pixels[0][2]);
AVG(tmp2, pixels[2][0], pixels[2][2]);
AVG(x_av, tmp1, tmp2);

#else
vert_av  = (pixels[0][1] + pixels[2][1])/2;
hori_av  = (pixels[1][0] + pixels[1][2])/2;
cross_av = (vert_av + hori_av)/2;
x_av     = ((pixels[0][0] + pixels[0][2])/2 + (pixels[2][0] + pixels[2][2])/2 )/2;
#endif


// Odd row
r_odd      =   (odd_col ? vert_av      : x_av);
r_even     =   (odd_col ? pixels[1][1] : hori_av);
r  =  odd_row ?  r_odd : r_even;


b_odd      =   (odd_col ? hori_av      : pixels[1][1]);
b_even     =   (odd_col ? x_av         : vert_av);
b  =  odd_row ?  b_odd : b_even;


g_odd      =   (odd_col ? pixels[1][1] : cross_av);
g_even     =   (odd_col ? cross_av     : pixels[1][1]);
g  =  odd_row ?  g_odd : g_even;



CLIPPING(r,clipping_val);
CLIPPING(b,clipping_val);
CLIPPING(g,clipping_val);


output_pixel->red = r;
output_pixel->green = g;
output_pixel->blue = b;

}


// ****************************************************************************
//
// This function is used to set the addresses of the line buffers for reading from and writing to
//
// ****************************************************************************
```

```
inline __attribute__ ((always_inline)) void rotate_line_delays()
{
   static unsigned int rotate_sbufs = 0; //NUM_LINES-1;
   unsigned int sbuf_bank_address_offset[NUM_LINES];
   unsigned int base_addr;
   const int LAST_BANK = ((NUM_LINES-1)*STREAM_BANK_SIZE);

   // first sbuf bank/line buffer start address... [0]
   // second sbuf bank/line buffer start address... [1]

   // @SK: Moving the increment to the end allows us to get rid of a jump

   base_addr = rotate_sbufs*STREAM_BANK_SIZE;
   sbuf_bank_address_offset[0] = base_addr;
   sbuf_bank_address_offset[1] = (sbuf_bank_address_offset[0] != LAST_BANK) ?
(sbuf_bank_address_offset[0])+STREAM_BANK_SIZE : 0;
   sbuf_bank_address_offset[2] = (sbuf_bank_address_offset[1] != LAST_BANK) ?
(sbuf_bank_address_offset[1])+STREAM_BANK_SIZE : 0;

   sbuf_set_read_address(0, sbuf_bank_address_offset[0]);
   sbuf_set_read_address(1, sbuf_bank_address_offset[1]);

   sbuf_set_write_address(0, sbuf_bank_address_offset[2]);  // will always write to the next sbuf which will hold the new
incoming line

   //rotate_sbufs = (rotate_sbufs != NUM_LINES-1) ? rotate_sbufs + 1 : 0;
//          ASM_COMMENT("fdsd 1");
   rotate_sbufs = (rotate_sbufs < (NUM_LINES-1)) ? rotate_sbufs + 1 : 0;
          // ASM_COMMENT("fdsd 2");

}

// ****************************************************************************
//
// This function creates the 3x3 shift-register.
// It takes a column of new data and shifts it to the start of the 2D array
//
// ****************************************************************************
inline __attribute__ ((always_inline)) void feed_block_3x3( unsigned int b[3][3], unsigned int c[3])
{
// Shift prev pixel in array
#define SHR(X,Y)  b[X][Y]=b[X][Y+1]
// Get new pixel from column
#define NEW(X,Y)  b[X][Y]=c[X]

   SHR(2,0); SHR(2,1); NEW(2,2);
   SHR(1,0); SHR(1,1); NEW(1,2);
   SHR(0,0); SHR(0,1); NEW(0,2);

#undef SHR
#undef NEW
}


/** rgb2yuv */
inline void rgb2yuv
(
   /* output */
   pixel_yuv_t* output,

   /* input */
   pixel_t input,

   enum e_yuv_formats yuv_format,

   int yuv_y_cof00,
   int yuv_y_cof01,
   int yuv_y_cof02,
   int yuv_yfloor,
```

```
    int yuv_cb_cof10,
    int yuv_cb_cof11,
    int yuv_cb_cof12,
    int yuv_cbfloor,

    int yuv_cr_cof20,
    int yuv_cr_cof21,
    int yuv_cr_cof22,
    int yuv_crfloor,

    int yuv_ceiling,
    int yuv_data_min_clip_en,
    int yuv_chroma_high_clip_en
)
{
    int y, cb, cr, tmp1, tmp2;
    int data_minval, chroma_clipval, high_th;


    // Need to delay the output etc to make the subsampling


    if (yuv_format == YUV422_121)
    {
      #define YUV_SHIFT 16       // The coefficents are multiplied by 1<<8
                                 // And the extra 1<<2 is to convert the
                                 // 10-bit RGB to 8-bit
      y = ((input.red   * yuv_y_cof00) >> YUV_SHIFT) +
          ((input.green * yuv_y_cof01) >> YUV_SHIFT) +
          ((input.blue  * yuv_y_cof02) >> YUV_SHIFT) +
          yuv_yfloor;

      cb= ((input.red   * yuv_cb_cof10) >> YUV_SHIFT)  +
          ((input.green * yuv_cb_cof11) >> YUV_SHIFT)  +
          ((input.blue  * yuv_cb_cof12) >> YUV_SHIFT)  +
          (yuv_cbfloor ) ;

      cr= ((input.red   * yuv_cr_cof20) >> YUV_SHIFT)  +
          ((input.green * yuv_cr_cof21) >> YUV_SHIFT)  +
          ((input.blue  * yuv_cr_cof22) >> YUV_SHIFT)  +
          (yuv_crfloor ) ;    // @SK: Not sure if this is correct
      #undef YUV_SHIFT


      //printf("y=%d cb=%d cr=%d\n", y, cb, cr);

      // @KF: Is this 255 right?? Is this a requirement for
      // YUV signals? In this case, we need to shift the result
      // found above by a bit to the right!

      high_th = 255 - yuv_ceiling;
      data_minval = (yuv_data_min_clip_en) ? 1 : 0;
      chroma_clipval = (yuv_chroma_high_clip_en) ? 254 : 255;


      y = (y <= 0) ? data_minval : y;
      y = (y > high_th) ? high_th : y;


      cb = (cb <= 0) ? data_minval : cb;
      cb = (cb > chroma_clipval) ? chroma_clipval : cb;


      cr = (cr <= 0) ? data_minval : cr;
      cr = (cr > chroma_clipval) ? chroma_clipval : cr;


      output->y = y;
      output->u = cb;
```

```
      output->v = cr;
   }
}

/** rgb2yuv */
inline void sub_sample_yuv422
(
   /* input */
   pixel_yuv_t  input,

   /* output */
   pixel_yuv_t* output,

   /* in-out*/
   pixel_yuv_t* delayed_yuv_1,
   pixel_yuv_t* delayed_yuv_2,

   int current_x,

   enum e_yuv_formats yuv_format

)
{
   int u, v;

   // The simplest way for now to do the submapling, is for output 0 of even
   // values (this value will not be stored anyway), and for the odd x's to
   // find the average over 3 pixels (previous, current and future).
   // --> Need a special way to compute the edges of the image


   if (yuv_format == YUV422_121)
   {

      output->y = delayed_yuv_1->y;
      // We output 0 when current_x & 1, since we are outputting the
      // previous
      u = ( delayed_yuv_2->u + 2*delayed_yuv_1->u + input.u ) / 4;

      // @SK: Normally we output 'u' when '(!(current_x & 1))' is true
      //      However for now, we will always output u, since the compiler
      //      is not intelligent enough to optimise this MUX out with the one
      //      that comes in ther upper level where either U or V is output

      /** output->u = (current_x & 1) ? 0 : u; **/
      output->u = u;


      v = ( delayed_yuv_2->v + 2*delayed_yuv_1->v + input.v ) / 4;
      // Read comment for the 'u' part'
      /** output->v = (!(current_x & 1)) ? 0 : v; **/
      output->v = v;

      delayed_yuv_2->y = delayed_yuv_1->y;
      delayed_yuv_2->u = delayed_yuv_1->u;
      delayed_yuv_2->v = delayed_yuv_1->v;

      delayed_yuv_1->y = input.y;
      delayed_yuv_1->u = input.u;
      delayed_yuv_1->v = input.v;

   }

}
```

D.2 FFT – C Source Code

The following is the C source code for the 8K radix-2 FFT used as a benchmark on DRAP and other architectures.

```
/*************************************************************
 * FFT_Radix-2.c
 *
 * Khodor Fawaz 2010.
 *
 * 8K radix-2 FFT.
 *
 *************************************************************/
#include <rica/rica.h>
#include <rica/source_sink.h>       // for read_source(), write_sink()
#include <rica/stream_buffers.h>
#include <stdio.h>
#include <stdlib.h>

int main (void)
{
   int i, j;
   int x1_im, x1_rel, x0_im, x0_rel, W_rel, W_im ;          // input pixel from the source cell
   int y0_im, y0_rel, y1_im, y1_rel, A, B;


   for(i=0; i<2056; i++)
   {
        //PIPELINE;              // This creates an asm comment which tells the scheduler to pipeline the kernel

        // read in the next incoming raw stream pixel
     //    input_pixel = read_source(0, true/*enable*/);
        x0_rel = read_source(0, true/*enable*/);
        x0_im = read_source(1, true/*enable*/);
        x1_rel = read_source(2, true/*enable*/);
        x1_im = read_source(3, true/*enable*/);


        W_rel = sbuf_read(0, true/*enable*/);
        W_im = sbuf_read(1, true/*enable*/);

        //output_pixel = input_pixel ;

                                    A = x1_rel*W_rel - x1_im*W_im;
                                    B = x1_im*W_rel + x1_rel*W_im;

                                    y0_im  = x0_im + B;
                                    y0_rel = x0_rel + A;
                                    y1_im  = x0_im - B;
                                    y1_rel = x0_rel - A;

//              output_pixel = input_pixel*7+43 + old_pixel;
//              old_pixel = input_pixel;

      write_sink_w_sync(0, y0_rel, 1, 1, 1);   //YUY2
            write_sink_w_sync(1, y0_im, 1, 1, 1);         //YUY2
            write_sink_w_sync(2, y1_rel, 1, 1, 1);        //YUY2
            write_sink_w_sync(3, y1_im, 1, 1, 1);         //YUY2
     //   output_16_bit(output_pixel, 1);
   }

   return 0;
}
```